Capitolo 2

Grafi e reti di flusso

Molti problemi di ottimizzazione sono caratterizzati da una struttura di grafo: in molti casi questa struttura emerge in modo naturale, in altri nasce dal particolare modo in cui i problemi vengono modellati. Ad esempio, una rete stradale è naturalmente rappresentabile come un grafo in cui i nodi sono gli incroci e gli archi le strade; pertanto non è strano che il settore dei trasporti sia uno di quelli in cui la teoria dei grafi trova maggiore applicazione. In molti altri problemi, invece, la struttura di grafo è più nascosta.

In questo capitolo studieremo alcuni problemi di base definiti su grafi e reti. Di essi daremo le proprietà più importanti e introdurremo alcuni algoritmi risolutivi, in generale i più semplici; per maggiori approfondimenti, si rinvia alla letteratura indicata ed a corsi avanzati dell'area di Ricerca Operativa. Tutti i problemi che tratteremo sono "facili", ossia ammettono algoritmi risolutivi di complessità polinomiale (e molto efficienti in pratica).

La conoscenza di algoritmi di base su grafi è un elemento importante nel curriculum di un esperto di informatica, anche al di fuori della Ricerca Operativa, in quanto strutture di grafo si incontrano sovente nel progetto e nella gestione di sistemi informatici, sia hardware che software: si pensi ad esempio alle reti di comunicazione (dalle WAN alle reti di interconnessione di calcolatori paralleli), ai compilatori, ai database, ecc.

2.1 Flussi su reti

In questo capitolo daremo per conosciuti i concetti elementari di teoria dei grafi riassunti nell'Appendice B, facendo specifici rinvii a tale appendice solo quando necessario. Introduciamo invece adesso alcuni concetti tipici dei problemi di ottimizzazione su grafo.

Con il termine "rete" indichiamo un grafo $G = (N, A)$ pesato, cioè ai cui nodi e/o archi sono associati valori numerici detti "pesi". In generale, in una rete gli archi sono interpretabili come canali attraverso cui fluiscono dei beni, che possono essere rappresentati per mezzo di grandezze discrete (ad esempio

il numero di auto su una strada, o il numero di messaggi su una rete di comunicazione) o continue (quantità di petrolio che fluisce in un oleodotto), possono rappresentare dei valori assoluti oppure dei valori relativi (per unità di tempo). In questo contesto, i pesi degli archi rappresentano usualmente delle capacità e dei costi, mentre i pesi dei nodi rappresentano la quantità dei beni che entrano nella rete, o che ne escono, in quei nodi. Più precisamente:

- ad ogni nodo $i \in N$ è associato un valore reale b_i, che può essere:
  - positivo, e in tal caso rappresenta la quantità del bene che esce dalla rete al nodo i; b_i è allora detto *domanda del nodo*, ed il nodo viene detto *destinazione*, *pozzo* o *nodo di output*;
  - negativo, e in tal caso rappresenta la quantità di bene che entra nella rete al nodo i; $-b_i$ è allora detto *offerta del nodo*, ed il nodo viene detto *origine*, *sorgente* o *nodo di input*;
  - nullo, ed in questo caso i viene detto *nodo di trasferimento*;
- ad ogni arco $a_k = (i, j)$ sono associati un *costo* c_k (o c_{ij}), che indica il costo che viene pagato per ogni unità del bene che attraversi l'arco, ed una *capacità inferiore* l_k (l_{ij}) e *superiore* u_k (u_{ij}), che indicano, rispettivamente, il minimo ed il massimo numero di unità di bene che possono attraversare l'arco. In molte applicazioni la capacità inferiore viene assunta uguale a 0, e quindi viene fornita tra i parametri della rete solamente la capacità superiore.

Nei problemi di flusso la domanda globale, cioè la somma di tutte le domande, è uguale all'offerta globale, cioè alla somma, cambiata di segno, di tutte le offerte; più formalmente, detti D e O, rispettivamente, gli insiemi dei nodi di domanda e di offerta:

$$D = \{i \in N : b_i > 0\}, \qquad O = \{i \in N : b_i < 0\},$$

si ha:

$$\sum_{i \in D} b_i = -\sum_{i \in O} b_i.$$

In altre parole, il vettore b, detto vettore dei *bilanci* dei nodi, deve soddisfare la relazione $\sum_{i \in N} b_i = 0$.

Data una rete $G = (N, A)$, con $|A| = m$, un vettore $x \in \mathbb{R}^m$ è un *flusso* su G se verifica i seguenti *vincoli di conservazione del flusso*:

$$\sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} = b_i, \qquad i \in N,$$

dove $BS(i)$ e $FS(i)$ sono rispettivamente la stella entrante e la stella uscente di $i \in N$, come definito nell'Appendice B. In forma vettoriale i vincoli di conservazione del flusso sono

$$Ex = b,$$

dove E è la matrice di incidenza del grafo e $b = [b_i]$.

Il valore x_k (o x_{ij}) è detto *flusso dell'arco* $a_k = (i, j)$. Un flusso è detto *ammissibile* se sono verificati i seguenti vincoli di capacità sugli archi:

$$l_{ij} \leq x_{ij} \leq u_{ij}, \qquad (i, j) \in A.$$

Tali vincoli possono essere scritti in forma vettoriale nel modo seguente:

$$l \leq x \leq u$$

dove $l = [l_{ij}]$ e $u = [u_{ij}]$.

Dato un flusso x, il costo del flusso è dato dalla somma dei flussi degli archi per il loro costo:

$$cx = \sum_{(i,j) \in A} c_{ij} x_{ij}.$$

Il seguente problema è detto *problema del flusso di costo minimo* (MCF, da *Min Cost Flow* problem):

$$\text{(MCF)} \qquad \begin{array}{ll} \text{Min} & cx \\ & Ex = b \\ & 0 \leq x \leq u \end{array} \tag{2.1}$$

In forma estesa, il problema diviene

$$\begin{array}{lll} \text{Min} & \displaystyle\sum_{(i,j) \in A} c_{ij} x_{ij} & \\ & \displaystyle\sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} = b_i & i \in N \\ & 0 \leq x_{ij} \leq u_{ij} & (i, j) \in A \end{array} \tag{2.2}$$

Esempio 2.1:

Sia dato il grafo orientato in figura 2.1, in cui sono riportati domande, offerte, costi e capacità superiori (si suppone che le capacità inferiori siano nulle).

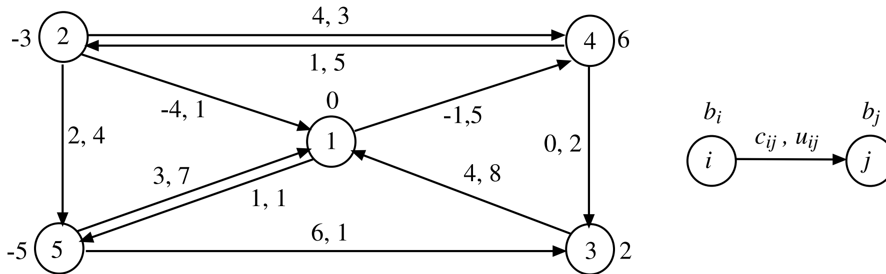

Figura 2.1: Un grafo con domande, offerte, costi e capacità

I vincoli di conservazione del flusso dei 5 nodi sono i seguenti:

$$
\begin{array}{ccccccccccccc}
-x_{14} & -x_{15} & +x_{21} & & & +x_{31} & & & +x_{51} & & = & 0 \\
 & & -x_{21} & -x_{24} & -x_{25} & & +x_{42} & & & & = & -3 \\
 & & & & & -x_{31} & & +x_{43} & & +x_{53} & = & 2 \\
+x_{14} & & & +x_{24} & & & -x_{42} & -x_{43} & & & = & 6 \\
 & +x_{15} & & & +x_{25} & & & & -x_{51} & -x_{53} & = & -5
\end{array}
$$

Si noti la forma della matrice di incidenza E. I vincoli di capacità degli archi sono i seguenti:

$$
\begin{array}{lll}
0 \leq x_{14} \leq 5 & 0 \leq x_{15} \leq 1 & 0 \leq x_{21} \leq 1 \\
0 \leq x_{24} \leq 3 & 0 \leq x_{25} \leq 4 & 0 \leq x_{31} \leq 8 \\
0 \leq x_{42} \leq 5 & 0 \leq x_{43} \leq 2 & 0 \leq x_{51} \leq 7 \\
0 \leq x_{53} \leq 1 & &
\end{array}
$$

Il costo del flusso è:

$$cx = -x_{14} + x_{15} - 4x_{21} + 4x_{24} + 2x_{25} + 4x_{31} + x_{42} + 3x_{51} + 6x_{53}.$$

Esercizio 2.1 *Determinare un flusso ammissibile per la rete in figura 2.1 e valutarne il costo.*

2.1.1 Alcuni modelli di flusso

Esiste un gran numero di problemi reali, in ambiti molto vari, che si modellano efficacemente come problemi di flusso: ne riportiamo di seguito alcuni esempi.

Schedulazione della produzione

L'industria dolciaria PereCani, nota produttrice di panettoni, deve decidere come utilizzare al meglio, nel corso dell'anno, la sua linea di produzione per tali dolci. La PereCani conosce una stima b_i, $i = 1, \ldots, 12$, del numero di panettoni che venderà in ogni mese dell'anno. Il costo unitario di produzione e la massima quantità di panettoni producibile in un mese variano anch'esse, a seconda di alcuni fattori quali il prezzo delle materie prime e la disponibilità di personale impegnato anche su altre linee di produzione, e anche di esse si hanno le stime, rispettivamente, c_i ed u_i, $i = 1, \ldots, 12$. I panettoni prodotti al mese i possono essere venduti immediatamente, oppure immagazzinati per essere poi venduti nei mesi successivi: il magazzino ha una capacità massima di U, ed un costo unitario di immagazzinamento pari a C. All'inzio il magazzino contiene b_0 panettoni, e si desidera che alla fine dell'anno ne contenga b_{13}.

Il problema della PereCani, noto in letteratura come problema di *Lot Sizing*, può essere formulato come un problema di flusso di costo minimo come mostrato in figura 2.2. Gli archi dal nodo fittizio 0 ai nodi 1, ..., 12 rappresentano la produzione, mentre gli archi di tipo $(i, i+1)$ rappresentano il magazzino. I bilanci ai nodi sono scelti in modo da rappresentare la vendita

di panettoni in ogni mese, al netto dei panettoni già presenti in magazzino (per il nodo 1) o di quelli che dovranno esservi lasciati (per il nodo 12); il bilancio al nodo 0 è la produzione totale di panettoni durante l'anno.

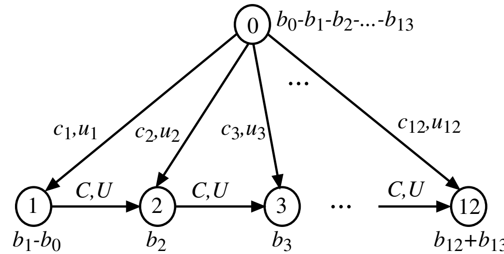

Figura 2.2: Il problema della PereCani

Schedulazione di viaggi di camion

La ditta di trasporti Andamiento Lento deve organizzare una giornata lavorativa per i suoi camion. La ditta deve effettuare n viaggi, ognuno caratterizzato da un tempo di inizio, una località di origine, un tempo di percorrenza ed una località di destinazione. I camion della ditta, tutti uguali, all'inizio della giornata sono tutti nello stesso deposito, e devono tutti trovarsi nel deposito alla fine della giornata. La località di origine del generico viaggio i, $i = 1, \ldots, n$, può essere raggiunta da un camion che parte dal deposito prima dell'istante di partenza del viaggio corrispondente, ed è noto il costo c_i^I (del carburante) che questo comporta; analogamente, è sempre possibile raggiungere il deposito dalla località di destinazione del viaggio i, $i = 1, \ldots, n$, prima della fine della giornata lavorativa, ed è noto il costo c_i^F che questo comporta. Una coppia di viaggi (i, j) è detta *compatibile* se essi possono essere effettuati, nell'ordine dato, dallo stesso camion; cioè se è possibile per il camion che ha effettuato il viaggio i, partire dalla località di destinazione di i e raggiungere la località di origine del viaggio j, prima del tempo di inizio di j. Per ogni coppia (i, j) di viaggi compatibile, è noto il costo c_{ij} del viaggio (a vuoto) tra la località di destinazione di i e la località di origine di j. Inoltre, esiste un costo fisso C che deve essere pagato per ogni camion che viaggia durante la giornata, indipendentemente dal numero di viaggi effettuati e dai km percorsi. Supponendo che la Andamiento Lento possegga abbastanza camion per eseguire tutti i viaggi (ovviamente non ne serviranno più di n), si vuole formulare il problema di scegliere quali viaggi far effettuare dallo stesso camion in modo da minimizzare il costo complessivo, dato dalla somma dei costi dei movimenti a vuoto (dal deposito alla prima origine, dalle destinazioni alle origini dei viaggi successivi e dalla destinazione dell'ultimo viaggio nuovamente al deposito) e dei costi fissi.

Questo problema può essere formulato come un problema di flusso di costo minimo nel seguente modo. Il grafo G ha $2n+2$ nodi: un nodo origine

s, un nodo destinazione t, e n coppie di nodi (i', i''), uno per ogni viaggio i, $i = 1, \ldots, n$. I nodi i' hanno bilancio 1, i nodi i'' hanno bilancio -1 mentre s e t hanno bilancio 0. Esistono archi (s, i') per ogni $i = 1, \ldots, n$, con costo c_i^I e capacità 1; analogamente, esistono archi (i'', t) per ogni $i = 1, \ldots, n$, con costo c_i^F e capacità 1. Per ogni coppia di viaggi (i, j) compatibile esiste un arco (i'', j') con costo c_{ij} e capacità 1. Infine, esiste un arco (t, s) con costo C e capacità infinita (anche se il flusso non può superare n). Un esempio di grafo per un problema con 5 viaggi è mostrato in figura 2.3, in cui il viaggio 1 è compatibile col 2 e col 3, il viaggio 2 è compatibile col 3 e col 5, il viaggio 4 è compatibile col 5 mentre i viaggi 3 e 5 non sono compatibili con nessun viaggio. Per semplicità, nella figura sono mostrati i bilanci ai nodi ma non i costi e le capacità degli archi.

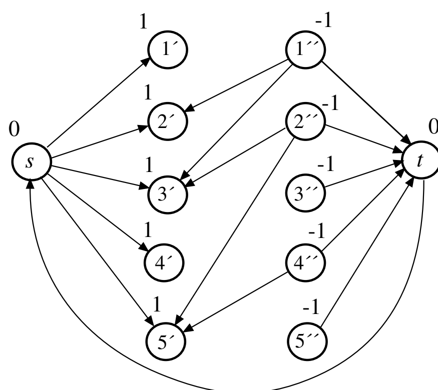

Figura 2.3: Il problema della Andamiento Lento

I nodi i' ed i'' rappresentano rispettivamente l'inizio e la fine del viaggio i, ed il flusso rappresenta i camion: un'unità di flusso su un arco (s, i') indica che il viaggio i viene effettuato da un camion appena uscito dal deposito, un'unità di flusso su un arco (i'', j') indica che i viaggi i e j vengono effettuati (in sequenza) dallo stesso un camion mentre un'unità di flusso su un arco (j'', t) indica che il camion che effettua il viaggio j torna immediatamente dopo al deposito. I vincoli di equilibrio ai nodi i' ed i'' garantiscono rispettivamente che il viaggio i sia compiuto da esattamente un camion (o proveniente dal deposito oppure a seguito di un altro viaggio) e che il camion che ha compiuto i torni al deposito oppure compia un altro viaggio. I vincoli di equilibrio ai nodi s e t garantiscono che tutti i camion che escono dal deposito vi rientrino; il numero di camion usati è pari al flusso sull'arco (t, s).

2.1.2 Trasformazioni equivalenti

Molti problemi reali possono essere formulati come problemi di flusso di costo minimo; questo è in parte dovuto alla grande flessibilità del modello

stesso. Infatti, sui problemi di (MCF) si possono fare alcune assunzioni che ne semplificano la descrizione e l'analisi: se tali assunzioni non sono rispettate dall'istanza che effettivamente occorre risolvere, è sempre possibile costruire un'istanza di (MCF), equivalente a quella data, che le rispetti. Tali assunzioni sono:

- *Singola sorgente - singolo pozzo*: se si hanno più sorgenti e/o pozzi, è possibile introdurre una *rete ampliata* $G' = (N', A')$ definita nel seguente modo:

  - $N' = N \cup \{s, t\}$,
  - $A' = A \cup \{(s, j) : j \in O\} \cup \{(i, t) : i \in D\}$,

  dove i nodi fittizi, s e t, sono la nuova sorgente e il nuovo pozzo. Ad ogni arco fittizio $(s, j) \in FS(s)$ viene associata una capacità $u_{sj} = -b_j$, uguale cioè al flusso in ingresso a $j \in O$ nel problema originario; analogamente, ad ogni arco fittizio $(i, t) \in BS(t)$ viene associata una capacità $u_{it} = b_i$, uguale cioè al flusso in uscita da $i \in D$ nel problema originario; tutti gli archi fittizi hanno costo 0. L'offerta di s e la domanda di t sono date da

  $$b'_s = \sum_{j \in O} b_j, \quad b'_t = \sum_{i \in D} b_i;$$

  mentre tutti gli altri nodi sono di trasferimento ($b'_i = 0, \forall i \neq s, t$). Un esempio di trasformazione della rete G nella rete ampliata G' è mostrata in figura 2.4.

  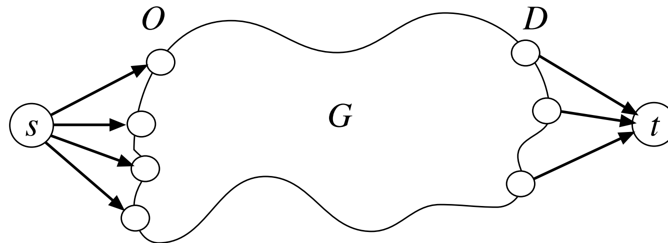

  Figura 2.4: La rete ampliata G'

  È facile dimostrare che ad ogni flusso ammissibile x' di G' corrisponde un flusso ammissibile x di G, e viceversa; infatti x' satura tutti gli archi (s, j) (e, di conseguenza, tutti gli archi (i, t)) e quindi le offerte e le domande ai nodi in O e in D sono rispettate.

- *Capacità inferiori nulle*: se un arco (i, j) ha capacità inferiore $l_{ij} \neq 0$, questo vuol dire che il flusso x_{ij} dovrà essere almeno pari a l_{ij}. Quindi, è possibile costruire un'istanza equivalente sottraendo l_{ij} a b_j e u_{ij},

sommando l_{ij} a b_i, ed aggiungendo un termine costante $c_{ij}l_{ij}$ alla funzione obiettivo: questo equivale a sostituire la variabile di flusso x_{ij} con $x'_{ij} = x_{ij} - l_{ij}$, e considerare l_{ij} unità di flusso permanentemente presenti sull'arco (i, j). È facile verificare che il problema della Andamiento Lento, descritto nel pragarafo precedente, è equivalente al problema di flusso di costo minimo in cui tutti i bilanci ai nodi sono nulli ed esistono archi (i', i'') con capacità sia inferiore che superiore pari a 1.

- *Nessuna capacità associata ai nodi*: in alcuni casi esiste una limitazione superiore u_i e/o una limitazione inferiore l_i alla massima quantità di flusso che può attraversare un nodo $i \in N$. Per questo, è sufficiente costruire un nuovo grafo G' in cui il nodo i è sostituito da due nodi i' ed i''. Tutti gli archi $(j, i) \in BS(i)$ vengono sostituiti con archi (j, i'), mentre gli archi $(i, j) \in FS(i)$ vengono sostituiti con archi (i'', j), con costi e capacità uguali agli archi originali; inoltre, viene aggiunto un arco (i', i'') con costo 0 e capacità superiore ed inferiore rispettivamente u_i e l_i. La domanda del nodo i viene attribuita ad i' se è positiva ed a i'' se negativa.

- *Eccesso di domanda o di offerta*: in alcuni problemi, il valore $-b_i$, $i \in O$, non rappresenta l'offerta al nodo i, ma la massima offerta che il nodo i può fornire alla rete. Se i valori b_j, $j \in D$, rappresentano la domanda di flusso dei nodi pozzo, il problema che si vuole risolvere è quello di determinare un flusso che soddisfi tutte le domande e per cui, per ogni nodo $i \in O$, l'offerta sia al più $-b_i$. Chiaramente, affinché esista un flusso ammissibile occorre che sia

$$-\sum_{i \in O} b_i \geq \sum_{i \in D} b_i \ .$$

Il problema può essere scritto come

$$\begin{array}{llll}
\min & \sum_{(i,j) \in A} c_{ij}x_{ij} & & \\
& \sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} & \geq b_i & \forall i \in O \\
& \sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} & = b_i & \forall i \notin O \\
& 0 \leq x_{ij} \leq u_{ij} & & \forall (i,j) \in A
\end{array}$$

È facile trasformare questo problema in un (MCF) ricorrendo ancora una volta ad una *rete ampliata* $G' = (N', A')$, avente un nuovo nodo sorgente s:

$$N' = N \cup \{s\}, \qquad A' = A \cup \{(s, i) : i \in O\},$$

dove, per ogni $i \in O$, l'arco (s, i) ha costo 0 e capacità $u_{si} = -b_i$. L'offerta della nuova sorgente s coincide con la domanda globale dei nodi pozzo:

$$b_s = -\sum_{j \in D} b_j;$$

inoltre si pone $b_i = 0$, per ogni $i \in O$, in modo che risulti $\sum_{i \in N'} b_i = 0$.

La trasformazione effettuata permette, una volta calcolati i flussi su G', di conoscere, per ciascuna sorgente $i \in O$, la capacità produttiva effettivamente utilizzata, x_{si}, e quella non utilizzata, $u_{si} - x_{si}$.

In modo analogo si tratta il caso in cui per ciascun nodo $i \in O$, è data l'offerta $-b_i$, mentre per ciascun nodo $j \in D$ è data una limitazione superiore b_j del valore che la domanda può assumere.

Nei prossimi paragrafi, considereremo reti con capacità inferiori nulle e senza capacità ai nodi. Talvolta ammetteremo casi in cui vi siano più sorgenti e/o pozzi.

Il problema di flusso di costo minimo è uno tra i più generali problemi di ottimizzazione su reti che ammettono algoritmi polinomiali: quasi tutti i problemi che studieremo nel resto del capitolo sono infatti casi particolari di (MCF). Questa generalità implica però che gli algoritmi risolutivi per i problemi di (MCF) sono tra i più complessi fra quelli di ottimizzazione su reti. Nel seguito introdurremo quindi prima problemi "più facili", per i quali si possono costruire algoritmi risolutivi relativamente semplici: tali algoritmi vengono utilizzati all'interno di algoritmi per il (MCF).

2.2 Visita di un grafo

Gli *algoritmi di visita* sono strumenti che consentono di individuare degli insiemi di nodi o delle porzioni di grafo che soddisfano particolari proprietà. Nel seguito descriveremo prima la versione base della procedura di visita, che risolve il problema di determinare, dato un grafo orientato $G = (N, A)$, l'insieme dei nodi raggiungibili per mezzo di un cammino orientato a partire da un dato nodo r. Mostreremo poi come tale versione di base possa essere utilizzata o adattata per risolvere problemi diversi.

La procedura *Visita* riceve in input il grafo orientato $G = (N, A)$ ed un nodo origine o radice r, e determina i nodi raggiungibili da r per mezzo di cammini orientati. Tali cammini individuano un albero orientato $T_r = (N_r, A_r)$ che viene fornito in output per mezzo di un vettore di predecessori $p[\cdot]$ (o funzione predecessore, si veda B.2.2). Al termine della procedura, ogni nodo i tale che $p[i] = 0$ non è stato raggiunto nella visita. Per il suo funzionamento la procedura si avvale di un insieme, Q, che contiene i *nodi candidati*, cioè quei nodi che sono già stati raggiunti nell'esplorazione ma ancora non sono stati utilizzati per proseguirla. Ad ogni passo la procedura

seleziona uno dei nodi in Q e prosegue la visita del grafo a partire da tale nodo; la correttezza dell'algoritmo non dipende dal modo in cui il nodo è selezionato, ossia dall'implementazione della funzione *Select*.

```
Procedure Visita (G, r, p):
  begin
    for i := 1 to n do p[i] := 0;
    p[r] := nil; Q := {r};
    repeat
      i := Select(Q); Q := Q \ {i};
      for each (i, j) ∈ FS(i) do
        if p[j] = 0 then begin p[j] := i; Q := Q ∪ {j} end
    until Q = ∅
  end.
```

Procedura 2.1: Visita di un grafo

Analisi della complessità. Ogni nodo i del grafo viene inserito in Q solamente la prima volta che viene raggiunto, quindi non si avranno più di n inserzioni e rimozioni di nodi da Q, e ogni arco (i, j) verrà esaminato al più una volta, se e quando i viene estratto da Q. Pertanto il numero globale di ripetizioni delle operazioni effettuate nel ciclo "**repeat** ... **until**" è limitato superiormente da m. Supponendo che le operazioni di gestione di Q abbiano complessità $O(1)$, si ha che la complessità di *Visita* è $O(m)$.

Esercizio 2.2 *Realizzare la procedura* Visita *per grafi memorizzati mediante liste di adiacenza.*

2.2.1 Implementazioni della procedura di visita

La correttezza della procedura di visita descritta nel paragrafo precedente è indipendente da:

- l'ordine con cui vengono esaminati gli archi della FS del nodo i estratto da Q, e
- l'ordine con cui vengono estratti i nodi da Q, ossia in che modo l'insieme viene implementato.

Questo significa che, indipendentemente dall'implementazione di queste due operazioni, si ottengono comunque tutti i nodi raggiungibili da r per mezzo di cammini orientati. Implementazioni diverse possono però fornire, al termine della procedura, insiemi di cammini diversi.

Non ci soffermeremo sull'effetto dell'ordinamento dei nodi nelle FS, che di solito, in pratica, dipende dai dettagli dell'implementazione delle strutture

dati con cui è rappresentato il grafo, e spesso, in ultima analisi, dall'ordine con cui sono descritti gli archi nell'input della procedura. Per semplicità, nel seguito assumeremo che le FS siano ordinate in senso crescente degli indici dei nodi testa.

Per quanto riguarda l'implementazione di Q, scelte diverse possono avere un impatto rilevante sull'insieme dei cammini individuati. In particolare, le implementazioni di Q come fila (*queue*) e pila (*stack*) corrispondono rispettivamente alle strategie di esplorazione del grafo note come *visita a ventaglio* (bfs, da *breadth-first search*) e *visita a scandaglio* (dfs, da *depth-first search*). Si noti che tutti i nodi inseriti in Q in quanto raggiungibili da uno stesso nodo i saranno "figli" di i nell'insieme di cammini determinato. In una visita *bfs*, tutti i figli di uno stesso nodo i vengono inseriti consecutivamente in Q, e quindi estratti consecutivamente da Q: di conseguenza, i discendenti di tali nodi possono essere estratti solamente dopo che l'ultimo di questi nodi è stato estratto, ossia i "fratelli" vengono visitati prima dei figli. In una visita *dfs*, i "figli" del nodo estratto i vengono inseriti in cima alla pila, e quindi saranno estratti (visitati) prima dei "fratelli" di i che sono ancora in Q al momento in cui i viene estratto.

Queste due strategie tendono a costruire insiemi di cammini con proprietà abbastanza diverse: in particolare, la visita a ventaglio (Q implementato come fila) tende a costruire cammini "corti", mentre la visita a scandaglio (Q implementato come pila) tende a costruire cammini "lunghi".

Esempio 2.2:
Applichiamo la procedura *Visita* al grafo in figura 2.5(a) partendo dal nodo $r = 1$.

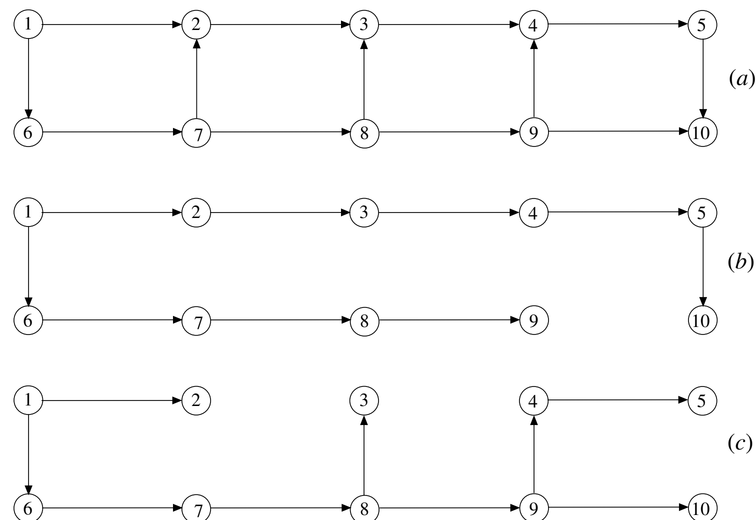

Figura 2.5: Applicazioni della procedura *Visita*

Se Q è implementato mediante una fila (*queue*), il risultato, in termini di cammini determinati, è mostrato in figura 2.5(b), L'ordine di inserzione in (e rimozione da) Q è:

$1, 2, 6, 3, 7, 4, 8, 5, 9, 10$. La rimozione di 1 da Q causa l'inserimento di 2 e 6 mediante gli archi $(1, 2)$ e $(1, 6)$; la rimozione di 2 causa l'inserimento di 3 mediante l'arco $(2, 3)$; la rimozione di 6 causa l'inserimento di 7 mediante l'arco $(6, 7)$, ecc. La funzione predecessore è definita dal vettore $p = [nil, 1, 2, 3, 4, 1, 6, 7, 8, 5]$. In questo caso il risultato è che tutti i nodi sono raggiungibili da r.

Diverso è il risultato, mostrato in figura 2.5(c), se Q è implementato mediante una pila (*stack*). In questo caso 2 e 6, "figli" di 1, vengono esaminati in ordine inverso (rispetto al caso precedente) in quanto 6 è inserito in Q dopo 2, e quindi ne è estratto prima. Inoltre, quando 6 viene rimosso da Q segue l'inserzione di 7 in cima alla pila, e quindi l'esplorazione prosegue da 7, "figlio" di 6, piuttosto che da suo "fratello" 2, e così via. Si noti come i cammini prodotti in questo caso possano essere diversi e più lunghi di quelli prodotti nel caso precedente; in particolare, nel caso (c) i cammini da 1 a 4 e da 1 a 5 sono formati rispettivamente da 5 e 6 archi, mentre nel caso (b) erano formati rispettivamente da 3 e 4 archi.

In effetti, è possibile dimostrare il seguente interessante risultato:

Teorema 2.1 *La procedura di visita in cui Q è implementato come fila determina, per ogni nodo i raggiungibile da r, un cammino orientato da r a i di lunghezza minima in termini di numero di archi.*

Dimostrazione Per ogni nodo i raggiungibile da r, denotiamo con $d(i)$ la "distanza" di i da r, ossia la lunghezza (in termini di numero di archi) del più corto cammino tra i ed r ($d(r) = 0$).

Dimostriamo per induzione che ogni nodo i con $d(i) = k > 0$ è inserito in Q – e quindi, siccome Q è una fila, estratto da Q – dopo tutti i nodi h con $d(h) = k - 1$ e prima di qualsiasi nodo j con $d(j) > k$ (se ne esistono); inoltre, al momento di essere inseriti in Q il loro predecessore $p[i] = h$ ha $d(h) = k - 1$ (ossia il cammino determinato dalla visita ha lunghezza pari a k). Questo è certamente vero per $k = 1$: un nodo i ha distanza 1 da r se e solo se $(r, i) \in FS(r)$, e tutti questi nodi sono esaminati e posti in Q alla prima iterazione della visita, quando viene esaminato r. Per il passo induttivo, supponiamo che la proprietà sia vera per k e dimostriamo che è vera per $k+1$. Dall'ipotesi induttiva segue immediatamente che quando il *primo* nodo i con $d(i) = k$ è estratto da Q, tutti i nodi h che erano stati estratti da Q in precedenza avevano $d(h) < k$: quindi, in quel momento tutti i nodi j con $d(j) > k$ non sono ancora stati inseriti in Q, e quindi hanno $p[j] = 0$. Un nodo j ha $d(j) = k+1$ se e solo se esiste almeno un nodo i con $d(i) = k$ tale che $(i, j) \in A$; siccome Q è una fila, dall'ipotesi induttiva segue che tutti i nodi con $d(j) = k + 1$ sono inseriti in coda alla fila nelle iterazioni della visita in cui sono estratti da Q ed esaminati tutti i nodi i con $d(i) = k$, e che il loro predecessore è uno di tali nodi. Da questo segue che la proprietà è vera anche per $k + 1$, e quindi il teorema. $\diamond$

Quindi, la procedura di visita è, in alcuni casi, in grado di calcolare insiemi di cammini che utilizzano il minimo numero di archi: nel paragrafo 2.3 vedremo come affrontare problemi di cammini minimi di tipo più generale.

2.2.2 Usi della procedura di visita

La procedura sopra descritta può essere modificata per risolvere anche altri problemi, tra i quali citiamo i seguenti:

- determinare l'insieme dei nodi raggiungibili per mezzo di un cammino orientato a partire da un dato *insieme* $R \subset N$ di nodi;
- determinare l'insieme dei nodi a partire dai quali un dato nodo r è raggiungibile per mezzo di un cammino orientato;
- determinare l'insieme dei nodi raggiungibili per mezzo di un cammino *non orientato* a partire da un dato nodo r, o, equivalentemente, determinare l'insieme dei nodi raggiungibili a partire da un dato nodo r su un grafo *non orientato*;
- individuare se un grafo è aciclico e, se non lo è, determinare un ciclo del grafo;
- determinare le componenti connesse di un grafo;
- determinare se un grafo è bipartito.

Tali problemi sono risolubili con piccole modifiche alla procedura *Visita*, e/o applicando la procedura ad un opportuno grafo ottenuto a partire da quello originario.

Ad esempio, supponiamo di voler determinare l'insieme dei nodi raggiungibili per mezzo di un cammino orientato a partire da un dato insieme R di nodi; è facile verificare che questo problema può essere risolto mediante un'applicazione della procedura *Visita* al grafo $G' = (N', A')$ in cui $N' = N \cup \{s\}$, dove s è un nodo fittizio che funge da "super-radice", e $A' = A \cup \{(s, r) : r \in R\}$.

Per il problema di determinare l'insieme dei nodi a partire dai quali r è raggiungibile per mezzo di un cammino orientato, invece, è sufficiente applicare la procedura *Visita*, con la stessa radice, al grafo $G' = (N, A')$ che ha gli stessi nodi del grafo originario G ma i cui archi sono "invertiti" rispetto a quelli di G, ossia tale che $A' = \{(j, i) : (i, j) \in A\}$. Analogamente, se si vuole determinare l'insieme dei nodi raggiungibili da r per mezzo di un cammino non orientato, è sufficiente applicare la procedura al grafo $G' = (N, A')$ in cui $A' = A \cup \{(j, i) : (i, j) \in A\}$.

Si noti che, negli esempi precedenti, è possibile evitare di costruire una rappresentazione del grafo G' modificando opportunamente la procedura *Visita* in modo che possa lavorare direttamente sulle strutture dati che descrivono il grafo originario G. Ad esempio, nel caso della determinazione dell'insieme dei nodi raggiungibili per mezzo di un cammino orientato a partire da un dato insieme R di nodi, è solamente necessario sostituire le istruzioni

$$p[r] := nil; \quad Q := \{r\};$$

con le istruzioni

$$\textbf{for each } r \in R \textbf{ do } p[r] := nil; \quad Q := R;$$

In altri termini, basta inizializzare tutti i nodi di R come nodi radice e porli tutti in Q all'inizio dell'algoritmo. Per il problema di determinare l'insieme dei nodi a partire dai quali r è raggiungibile per mezzo di un cammino orientato, invece, è sufficiente modificare l'istruzione

for each $(i, j) \in FS(i)$ **do**

in

for each $(j, i) \in BS(i)$ **do**

Analogamente, per il problema della raggiungibilità attraverso cammini non orientati è sufficiente esaminare sia gli archi $(i, j) \in FS(i)$ che gli archi $(j, i) \in BS(i)$ corrispondenti al nodo i estratto da Q.

Esercizio 2.3 *Ovviamente, le operazioni precedenti si possono combinare: ad esempio, si discuta come modificare la procedura di visita per determinare l'insieme di nodi a partire dai quali almeno uno dei nodi in $R \subset N$ è raggiungibile mediante un cammino orientato.*

Altri problemi possono essere risolti con l'uso ripetuto della procedura di visita o con modifiche minori alla stessa. Alcuni di questi problemi sono descritti nei seguenti esercizi.

Esercizio 2.4 *Si proponga un algoritmo di complessità $O(m)$, basato sulla procedura di visita, che determini il numero di componenti connesse di un grafo non orientato, fornendone lo pseudo-codice; si noti che in un grafo non orientato $FS(i)$ e $BS(i)$ non sono definite, è definita solamente la stella $S(i)$ degli archi incidenti nel nodo i (suggerimento: la visita a partire da un qualunque nodo i determina la compenente connessa di cui i fa parte; al termine della visita i nodi delle altre componenti connesse hanno predecessore nullo).*

Esercizio 2.5 *Si modifichi l'algoritmo dell'esercizio precedente in modo tale che, con la stessa complessità, produca un vettore $cc[\cdot]$ tale che $cc[i] = k$ se e solo se il nodo i appartiene alla k-esima componente connessa.*

Esercizio 2.6 *Si costruisca una versione modificata della procedura di visita (fornendo lo in pseudo-codice) che risolva il problema di determinare se un dato grafo non orientato e connesso sia aciclico, ossia se contenga oppure no cicli. Nel caso che il grafo non sia aciclico, si richiede che la procedura produca in output (come "certificato") un ciclo del grafo. Si discuta la complessità di tale procedura.*

Esercizio 2.7 *Si adatti l'algoritmo dell'esercizio precedente al caso di un grafo non connesso. Si discuta la complessità di tali procedure.*

Esercizio 2.8 *Fornire un algoritmo di visita, di complessità $O(m)$, per verificare se un dato grafo non orientato, eventualmente non connesso, sia bipartito.*

Esercizio 2.9 *Fornire un algoritmo di visita, di complessità $O(m)$, per verificare se un grafo, orientato e connesso, sia fortemente connesso (suggerimento: è sufficiente verificare che un arbitrario nodo r del grafo è connesso a tutti gli altri nodi e questi sono connessi ad r mediante cammini orientati).*

2.3 Cammini di costo minimo

Il problema della determinazione di cammini di costo minimo, detti anche *cammini minimi*, è uno tra i più semplici, ma allo stesso tempo tra i più importanti problemi di ottimizzazione su reti. Lo illustriamo attraverso alcuni esempi.

Esempio 2.3: Visite pedonali di Pisa
Si vuole installare alla stazione ferroviaria di Pisa e in un paio di parcheggi periferici della città dei "totem", cioè dei calcolatori con "touch screen", attraverso i quali poter interrogare un GIS (Geographical Information System) per determinare i percorsi a piedi più corti dal luogo in cui ci si trova verso le possibili destinazioni turistiche interne alla città. Altri "totem" verranno installati nelle piazze turistiche per permettere di selezionare nuovi percorsi.

Nel GIS, la città è stata completamente descritta mediante un grafo orientato $G = (N, A)$, in cui ogni arco rappresenta un tratto di percorso a piedi (strada, vicolo, ponte, attraversamento di una grande piazza, ecc.) e i nodi non sono altro che i punti di incontro di tali archi. Ad ogni arco è associato un "costo", cioè un valore reale che rappresenta la sua lunghezza.

L'utente introduce la propria destinazione e un programma nel "totem" calcola il percorso di minima lunghezza dall'origine alla destinazione indicata fornendo tutti i dettagli del cammino sullo schermo in forma grafica e, a richiesta, stampando una lista di informazioni per seguire agevolmente il percorso.

Mentre nell'esempio appena presentato la formulazione in termini di problema di cammini minimi è una naturale e diretta conseguenza delle caratteristiche del problema, il prossimo esempio mostra come un problema di cammini minimi può servire a formulare e risolvere problemi che apparentemente non hanno alcun rapporto con i grafi e le reti.

Esempio 2.4: Ispezioni su una linea di produzione
Si abbia una linea di produzione con n celle di lavorazione. Ogni lotto è costituito da B pezzi che passano attraverso le n celle ed in ciascuna di esse subiscono una lavorazione. La probabilità di produrre un difetto in un pezzo nella cella i è p_i. Possono essere fatte ispezioni alla fine di ogni lavorazione: le ispezioni vengono fatte su tutti i pezzi del lotto e quelli trovati difettosi vengono scartati. Non essendo accettabile l'invio ai clienti di pezzi difettosi, viene comunque fatta una ispezione alla fine; tuttavia può essere conveniente effettuare ispezioni già dopo le prime lavorazioni in modo da evitare il costo di lavorazioni effettuate su pezzi difettosi e quindi da scartare. Sia:

q_i il costo unitario di lavorazione alla cella i;

f_{ij} il costo fisso di ispezione di un lotto all'uscita della cella j, nell'ipotesi che la precedente ispezione era stata effettuata all'uscita della cella $i(< j)$;

h_{ij} il costo unitario di una ispezione effettuata all'uscita della cella j, nell'ipotesi che la precedente ispezione era stata effettuata all'uscita della cella $i(< j)$.

Il numero atteso di pezzi non difettosi alla fine della lavorazione i è dato da:

$$B_i = B \prod_{k=1}^{i} (1 - p_k);$$

B_i è il numero di pezzi su cui si effettueranno le lavorazioni nelle celle successive alla cella i, sino a quella in cui si effettuerà una nuova ispezione. Il costo di un'ispezione effettuata alla cella j nell'ipotesi che la precedente sia stata effettuata alla cella $i(< j)$ è dato da $f_{ij} + B_i h_{ij}$; sommando ad esso il costo di lavorazione dei pezzi in tutte le celle da $i+1$ a j comprese, si ottiene il costo globale (produzione e ispezione) nel segmento produttivo da i escluso a j compreso:

$$c_{ij} = f_{ij} + B_i h_{ij} + B_i \sum_{k=i+1}^{j} q_k.$$

Nel seguito con 0 indicheremo una cella fittizia, che precede l'inizio del processo produttivo, affinché siano definiti i valori f_{0j}, h_{0j}, c_{0j} e $B_0 = B$ relativi al caso in cui la prima ispezione sia effettuata nella cella j.

Il problema di determinare il piano di ispezioni ottimo, cioè decidere quando effettuare le ispezioni in modo da minimizzare il costo globale (la somma dei costi di produzione e di quelli di ispezione), può essere formulato come il problema di cercare un cammino di costo minimo dal nodo 0 al nodo n, nel grafo $G = (N, A)$, con $N = \{0, 1, \ldots, n\}$, e $A = \{(i, j) : i = 0, 1, \ldots, n-1; j = i+1, \ldots, n\}$, in cui ad ogni arco (i, j) è associato il costo c_{ij}. In figura 2.6 è mostrato il grafo nel caso di $n = 4$.

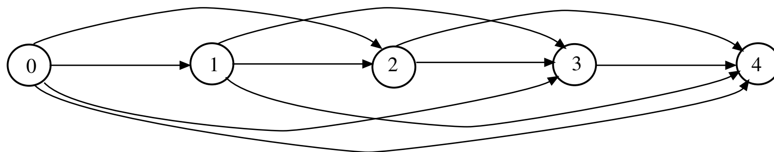

Figura 2.6: Grafo associato al problema delle ispezioni

È facile dimostrare che ogni cammino orientato del grafo da 0 a n corrisponde a un piano di ispezione e, viceversa, qualunque piano di ispezione è rappresentato da un cammino orientato da 0 a n, il cui costo (come somma dei costi degli archi) è proprio il costo globale di produzione e ispezione.

Nel seguito daremo prima una definizione formale del problema dei cammini minimi per presentare, poi, i principali algoritmi per la sua risoluzione.

2.3.1 Il problema

Sia $G = (N, A)$ un grafo orientato e pesato dove ad ogni arco $(i, j) \in A$ è associato un costo $c_{ij} \in \mathbb{R}$. Per ogni cammino P in G, il costo $C(P)$ è dato dalla somma dei costi degli archi che lo costituiscono:

$$C(P) = \sum_{(i,j) \in P} c_{ij} \ .$$

Dati due nodi r e t, definiamo $\mathcal{P}_{rt}$ l'insieme dei cammini (orientati) che connettono r a t; il *problema del cammino minimo da r a t* è

$$\min\{C(P) : P \in \mathcal{P}_{rt}\}. \tag{2.3}$$

Il problema (2.3) può essere formulato come un particolare problema di flusso di costo minimo su G, in cui gli archi hanno capacità infinita e i costi sono quelli del problema del cammino minimo, r è l'unica sorgente e produce 1 unità di flusso, t è l'unico pozzo e richiede 1 unità di flusso, mentre ogni altro nodo è di trasferimento:

$$\begin{array}{lrcl} \text{Min} & cx & & \\ & Ex & = & b \\ & x & \geq & 0, \quad x \text{ intero} \end{array} \tag{2.4}$$

con

$$b_i = \begin{cases} -1, & \text{se } i = r, \\ 1, & \text{se } i = t, \\ 0, & \text{altrimenti.} \end{cases}$$

Infatti, il modo migliore per soddisfare la richiesta di un'unità di flusso da parte di t è inviarla lungo un cammino di costo minimo da r a t. Si noti che, se non si fosse imposto il vincolo di interezza su x, nel caso in cui esistano più cammini aventi uguale lunghezza minima si sarebbero potute inviare frazioni dell'unità di flusso su cammini diversi.

È possibile considerare un problema più generale rispetto a (2.3): data una radice r, determinare in G un cammino di costo minimo da r a i, per ogni $i \neq r$. È facile vedere che tale problema può essere formulato come

$$\min\{ \Sigma_{i \neq r} C(P_i) : P_i \in \mathcal{P}_{ri}, i \neq r \}. \tag{2.5}$$

Infatti, la scelta del cammino per un dato nodo t non influenza la scelta del cammino per tutti gli altri nodi; quindi il modo migliore per minimizzare la somma dei costi di tutti i cammini è quella di selezionare per ogni nodo il cammino di costo minimo. Il problema (2.5) può essere formulato come un problema di flusso di costo minimo su G in modo analogo a (2.4), con l'unica differenza che la radice r è la sorgente di $n-1$ unità di flusso ($n = |N|$), mentre ogni altro nodo i è un nodo pozzo e richiede un'unità di flusso, ossia

$$b_i = \begin{cases} -(n-1), & \text{se } i = r, \\ 1, & \text{altrimenti.} \end{cases}$$

Dato che la funzione obiettivo è la *somma* delle lunghezze di tutti i cammini P_i, $i \neq r$, se nella soluzione ottima un certo arco (i, j) è contenuto in k distinti cammini di costo minimo da r a k diversi nodi, si avrà $x_{ij} = k$, ossia il costo di ciascun arco viene conteggiato una volta per ogni cammino di costo minimo di cui fa parte.

Il motivo per cui usualmente si considera (2.5) invece di (2.3) è che, nel caso peggiore, anche per la determinazione di un cammino minimo per un solo nodo destinazione è necessario determinare tutti gli altri. Inoltre, in molte applicazioni si debbono calcolare i cammini minimi aventi un'origine comune e destinazione negli altri nodi.

Osserviamo che, senza perdere di generalità, si può assumere che esista almeno un cammino da r a ciascun altro nodo i: infatti, è sempre possibile aggiungere un arco fittizio (r, i), per ogni $i \neq r$ tale che $(r, i) \notin A$, con costo $c_{ri} = M$, dove M è un numero molto grande (è sufficiente che sia $M \geq (n-1)c_{max} + 1$, dove c_{max} è il massimo dei costi degli archi). Il cammino di costo minimo sarà costituito dal solo arco fittizio (r, i) solo se non esiste alcun cammino da r a i: infatti, se tale cammino esistesse il suo costo sarebbe certamente inferiore a M.

È possibile che il problema (2.5) sia inferiormente illimitato: questo accade se e solo se nel grafo esiste un *ciclo negativo*, cioè un ciclo orientato il cui costo sia negativo. Infatti, supponiamo che esista un ciclo negativo e consideriamo un cammino che lo tocchi in un nodo: percorrendo il ciclo si ottiene un cammino non semplice, e più volte si percorre il ciclo più si riduce il costo del cammino, mostrando così che il problema non è inferiormente limitato. Se il grafo non contiene cicli negativi, rimuovendo i cicli da cammini non semplici si ottengono cammini semplici non più costosi, pertanto esistono soluzioni ottime formate da cammini semplici.

Nel seguito supporremo che il grafo sia privo di cicli negativi, e quindi che (2.5) abbia soluzione ottima finita $\{P_i, i \neq r\}$; vedremo inoltre che è possibile verificare in tempo polinomiale se un grafo contiene un ciclo negativo. È facile verificare che, tra tutte le soluzioni ottime di (2.5), ne esiste almeno una in cui *l'unione dei cammini P_i forma un albero di copertura per G radicato in r e orientato*, ossia un albero di radice r i cui archi sono orientati da r verso le foglie. Infatti, se P_i è un cammino minimo da r a i e j è un nodo interno al cammino, allora il sottocammino di P_i che arriva sino a j è a sua volta un cammino minimo da r a j. Quindi, se esistono più cammini minimi da r ad un certo nodo i, è possibile selezionarne uno, P_i, ed imporre che i cammini minimi, da r ad altri nodi, che passano per i abbiano P_i come sottocammino da r ad i.

Ogni soluzione ottima di (2.5) che possa essere rappresentata mediante un albero di copertura orientato di radice r è detta un *albero di cammini minimi di radice r*. Nel seguito considereremo la seguente forma equivalente di (2.5): determinare in G un *albero di cammini minimi* di radice r. Questo viene detto *problema dell'albero di cammini minimi* (SPT, da Shortest Path Tree), o, più semplicemente, *problema dei cammini minimi.*

2.3.2 Alberi, etichette e condizioni di ottimo

Sia $T = (N, A_T)$ una soluzione ammissibile per (SPT), ossia un albero di copertura radicato in r e orientato; verifichiamo se T è una soluzione ottima. Dobbiamo cioè verificare se, per qualche nodo $i \neq r$, esiste un cammino orientato da r ad i di costo minore di $C(P_i^T)$, dove P_i^T è l'unico cammino da r ad i in T. Per fare questo calcoliamo il costo dei cammini in T: costruiamo quindi un vettore di *etichette dei nodi* $d \in \mathbb{R}^n$ tale che $d_i = C(P_i^T)$, per $i \neq r$,

e $d_r = 0$. Il vettore d può essere facilmente determinato per mezzo di una procedura di visita dell'albero a partire dalla radice r. Si noti che, se l'albero contiene l'arco (i, j), allora $d_i + c_{ij} = d_j$; infatti, l'unico cammino da r a j è formato dal sottocammino da r ad i, di costo d_i, e dall'arco (i, j), che ha costo c_{ij}.

Esercizio 2.10 *Si particolarizzi la procedura* Visita *in modo che, dato un albero T con costi sugli archi, calcoli il vettore di etichette d.*

Dato il vettore delle etichette d corrispondente a T, è possibile verificare se qualche arco $(i, j) \notin A_T$ può essere utilizzato per costruire un cammino da r a j migliore di P_j^T. Infatti, supponiamo che per un qualche arco (i, j) risulti $d_i + c_{ij} < d_j$, e sia h il predecessore di j in T (il nodo immediatamente precedente j nel cammino P_j^T): sostituendo nell'albero l'arco (h, j) con (i, j) si ottiene un nuovo albero T' in cui il nodo j è raggiunto con un cammino di costo inferiore, come mostrato in figura 2.7.

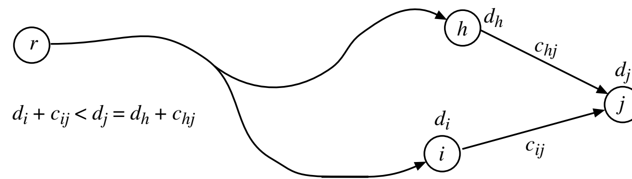

Figura 2.7: individuazione di un cammino di costo inferiore per j

Se invece ciò non accade per alcun arco, allora T è una soluzione ottima per (SPT); per dimostrarlo, utilizziamo il seguente lemma:

Lemma 2.1 *Sia $d \in \mathbb{R}^n$ un vettore di etichette dei nodi che verifica le seguenti* "condizioni di Bellman"

$$d_i + c_{ij} \geq d_j, \quad \forall (i, j) \in A \tag{2.6}$$

e tale che $d_r = 0$; allora, per ogni $i \neq r$, d_i è una valutazione inferiore del costo del cammino minimo da r a i.

Dimostrazione Sia $P_i = \{j_1, j_2, \ldots, j_k\}$ un qualsiasi cammino, non necessariamente semplice, da r a i (quindi $r = j_1$ e $j_k = i$). Per ipotesi si ha:

$$\begin{array}{lcl} d_{j_k} & \leq & d_{j_{k-1}} + c_{j_{k-1} j_k} \\ d_{j_{k-1}} & \leq & d_{j_{k-2}} + c_{j_{k-2} j_{k-1}} \\ \vdots & & \vdots \\ d_{j_2} & \leq & d_{j_1} + c_{j_1 j_2} \end{array} .$$

Sommando membro a membro, e sapendo che $d_{j_1} = d_r = 0$, si ottiene:

$$d_{j_k} = d_i \leq C(P_i) = \sum_{\ell=1}^{k-1} c_{j_\ell j_{\ell+1}},$$

il che, essendo vero per ogni cammino $P_i \in \mathcal{P}_{ri}$, è vero in particolare per il cammino di costo minimo. $\diamond$

Quindi, se il vettore di etichette d corrispondente a T verifica (2.6), allora T è chiaramente ottimo: per il Lemma 2.1, il cammino minimo da r ad un qualsiasi nodo $i \neq r$ non può costare meno di d_i, ed il cammino P_i^T ha esattamente quel costo. Se invece il vettore di etichette d corrispondente a T non verifica (2.6) allora, come abbiamo visto (cf. figura 2.7), T non può essere ottimo. Si ottiene quindi il seguente risultato:

Teorema 2.2 *Sia $T = (N, A_T)$ un albero di copertura radicato in r e orientato, e sia d il corrispondente vettore di etichette: T è un albero dei cammini minimi di radice r se e solo se d verifica le condizioni di Bellman (2.6).*

Si noti che, per verificare l'ottimalità di T, abbiamo associato un'etichetta ad ogni nodo: anche se fossimo interessati solamente al cammino minimo da r ad un dato nodo t, per dimostrarne l'ottimalità attraverso le condizioni di Bellman dovremmo comunque associare un'opportuna etichetta anche a tutti gli altri nodi. Inoltre, un vettore di etichette che rispetta le condizioni di Bellman fornisce informazione sul costo del cammino minimo da r ad ogni altro nodo; questo mostra perché usualmente si consideri il problema dell'albero dei cammini minimi.

È interessante notare che, se nel grafo esiste un ciclo orientato di costo negativo, allora non esiste (almeno per alcuni nodi del grafo) nessun limite inferiore al costo dei cammini minimi, e quindi non può esistere nessun vettore di etichette che rispetta le condizioni di Bellman. Infatti, sia $P = \{j_1, j_2, \ldots, j_k, j_1\}$ un ciclo orientato di costo negativo, e supponiamo per assurdo che esista un vettore d che rispetta le condizioni di Bellman: si ha

$$\begin{array}{lcl} d_{j_k} & \leq & d_{j_{k-1}} + c_{j_{k-1}j_k} \\ d_{j_{k-1}} & \leq & d_{j_{k-2}} + c_{j_{k-2}j_{k-1}} \\ \vdots & & \vdots \\ d_{j_2} & \leq & d_{j_1} + c_{j_1 j_2} \\ d_{j_1} & \leq & d_{j_k} + c_{j_k j_1} \end{array}$$

da cui, sommando membro a membro, si ottiene la contraddizione $0 \leq C(P) < 0$.

2.3.3 L'algoritmo *SPT*

Le condizioni di ottimalità viste nel precedente paragrafo suggeriscono in modo naturale il seguente algoritmo per la determinazione di un albero dei cammini minimi:

- mediante una visita del grafo si determina un albero di copertura radicato in r ed orientato (rappresentato dal vettore $p[\cdot]$) e le relative

etichette d_i, che rappresentano il costo dell'unico cammino sull'albero da r a i;

- si controlla se esiste un arco $(i, j) \in A$ tale che $d_i + c_{ij} < d_j$; in caso affermativo si modifica l'albero togliendo l'arco $(p[j], j)$ e sostituendovi l'arco (i, j) (ponendo $p[j] = i$), si calcola il vettore delle etichette corrispondente al nuovo albero e si itera; altrimenti l'algoritmo termina avendo dimostrato che l'albero corrente è ottimo (sono verificate le condizioni di Bellman).

Questo algoritmo è un algoritmo di *ricerca locale* (si veda l'Appendice A) in cui l'intorno è rappresentato dall'insieme di tutti gli alberi di copertura radicati in r ed orientati che differiscono da quello corrente per al più un arco. È facile vedere che il ricalcolo delle etichette a seguito della sostituzione di $(p[j], j)$ con (i, j) va effettuato solo per i nodi nel sottoalbero di radice j, in cui tutte le etichette diminuiscono della stessa quantità $d_j - d_i - c_{ij} > 0$.

Per ottenere un algoritmo più efficiente è possibile differire l'aggiornamento delle etichette. L'algoritmo mantiene ad ogni iterazione una soluzione ammissibile, rappresentata da un vettore di predecessori $p[\cdot]$, una struttura, che indicheremo con Q, che contiene tutti i nodi i cui archi uscenti potrebbero violare le condizioni di Bellman (2.6), ed un vettore di etichette $d[\cdot]$ in cui $d[i]$, in questo caso, rappresenta in generale un'*approssimazione superiore* del costo dell'unico cammino sull'albero da r a i: all'inizio, l'albero è formato da archi fittizi (r, i) aventi costo M molto grande ($p[i] = r$ e $d[i] = M$, per $i \neq r$).

```
Procedure SPT (r, p, d):
  begin
    for i := 1 to n do begin p[i] := r; d[i] := M end;
    p[r] := nil; d[r] := 0; Q := {r};
    repeat
      select u from Q; Q := Q \ {u};
      for each (u, v) ∈ FS(u) do
        if d[u] + c[u, v] < d[v] then
          begin
            d[v] := d[u] + c[u, v]; p[v] := u;
            if v ∉ Q then Q := Q ∪ {v}
          end
    until Q = ∅
  end.
```

Procedura 2.2: Algoritmo *SPT*

L'algoritmo, che chiameremo *SPT*, controlla se le condizioni di Bellman sono verificate, e ogni volta che trova un arco (i, j) per cui esse sono violate, cioè per cui $d_i + c_{ij} < d_j$, modifica la coppia (p_j, d_j) ponendo $p_j = i$ e $d_j = d_i + c_{ij}$. A seguito della diminuzione di d_j, tutti gli archi uscenti da j possono violare le condizioni di Bellman; j viene detto per questo *nodo candidato*, e viene inserito in Q. Ad ogni iterazione si verifica se Q è vuoto: in questo caso l'algoritmo termina avendo determinato una soluzione ottima, infatti il vettore $d[\cdot]$ rispetta le condizioni di Bellman e contiene le etichette dell'albero T rappresentato dal vettore $p[\cdot]$. Altrimenti, si estrae un nodo u da Q e si controlla se le condizioni (2.6) valgono per ciascun arco della sua stella uscente. Per ogni arco (u, v) che non soddisfa le condizioni, si pone $p[v] = u$ e l'etichetta di v viene aggiornata. Non si effettua però l'aggiornamento delle etichette in tutti i nodi del sottoalbero di radice v, ma si inserisce v in Q: si effettua cioè un "aggiornamento differito" delle etichette.

Esempio 2.5:
Si voglia determinare un albero dei cammini minimi di radice $r = 1$ sul grafo in figura 2.8(a), applicando l'algoritmo *SPT* in cui Q è implementato come una fila.

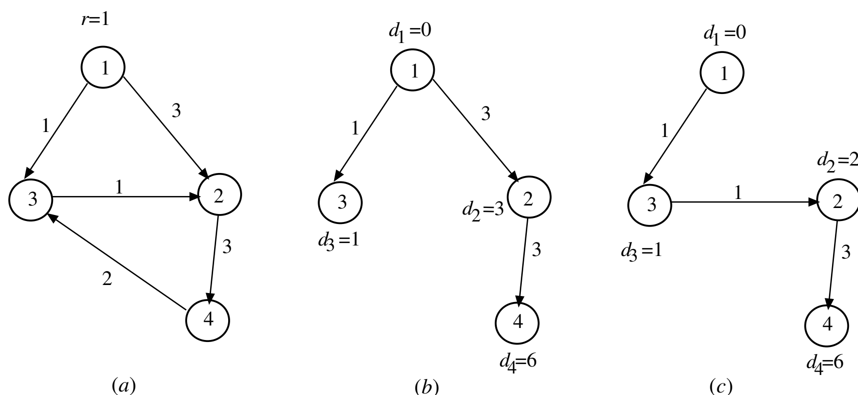

Figura 2.8: Un passo della procedura *SPT*

Esaminando la stella uscente del nodo 1 si pone $p[2] := 1$, $d_2 := 3$, $p[3] := 1$ e $d_3 := 1$; al termine della prima iterazione si ha dunque $Q = \{2, 3\}$. Viene quindi estratto 2 da Q, ed esaminando la sua stella uscente si pone $p[4] := 2$ e $d_4 := 6$, come mostrato in figura 2.8(b), con $Q = \{3, 4\}$. A questo punto si seleziona 3 da Q e si controlla la condizione di Bellman per l'arco $(3, 2)$: $d_3 + c_{32} = 1 + 1 < 3 = d_2$. Si modificano pertanto l'etichetta ($d_2 := d_3 + c_{32} = 2$) e il predecessore del nodo 2 ($p[2] := 3$) e si inserisce nuovamente 2 in Q ottenendo l'albero di figura 2.8(c), con $Q = \{4, 2\}$. Osserviamo che l'etichetta del nodo 4 non viene modificata, e quindi non rappresenta il costo del cammino da 1 a 4 nell'albero rappresentato dal vettore $p[\cdot]$. Però, quando, dopo aver estratto da Q il nodo 4 (senza causare nessun cambiamento), verrà selezionato il nodo 2 e controllato l'arco $(2, 4)$, d_4 verrà aggiornato e posto al valore 5, diventando così uguale al costo del cammino da 1 a 4 sull'albero e 4 verrà inserito in Q. Questo è l'aggiornamento differito delle etichette del sottoalbero. L'algoritmo termina con la seconda estrazione di 4 senza alcun aggiornamento di etichette.

Per dimostrare la terminazione dell'algoritmo *SPT* abbiamo bisogno del seguente risultato:

Teorema 2.3 *Ad ogni passo della procedura* SPT*, il valore d_v dell'etichetta del nodo v, per ogni $v \in N$, rappresenta il costo di un cammino da r a v nel grafo G (oppure è M).*

Dimostrazione La dimostrazione è per induzione sul numero delle iterazioni. La tesi è certamente vera alla prima iterazione, subito dopo la fase di inizializzazione. Assumendo che la tesi sia vera all'iterazione k, verifichiamo che lo sia anche all'iterazione $k+1$. Sia j un nodo la cui etichetta è migliorata all'iterazione $k+1$: il nuovo valore sarà $d_j = d_i + c_{ij}$ per qualche nodo i. Poiché l'etichetta di i è stata modificata in un'iterazione precedente, per ipotesi induttiva essa rappresenta il costo di un cammino da r ad i. Pertanto d_j è il costo di un cammino costituito da un sottocammino da r ad i, di costo d_i, e dall'arco (i, j). ◇

A questo punto possiamo dimostrare che, se il grafo non contiene cicli di costo negativo, la procedura *SPT* termina dopo un numero finito di passi. Infatti, per il Teorema 2.3 il valore d_i dell'etichetta di qualsiasi nodo i è sempre uguale al costo di un cammino del grafo G da r ad i. Osserviamo che il nodo i è inserito in Q solo quando la sua etichetta diminuisce: poiché il numero di cammini semplici da r ad i è finito, d_i può diminuire solamente un numero finito di volte, e quindi il nodo i potrà essere inserito in Q solamente un numero finito di volte. Di conseguenza, dopo un numero finito di iterazioni si avrà $Q = \emptyset$ e la procedura terminerà.

Il prossimo esempio mostra che, se invece il grafo contiene un ciclo negativo (e almeno uno dei nodi del ciclo è raggiungibile da r), allora la procedura *SPT non termina.*

Esempio 2.6:
Si consideri nuovamente il problema di figura 2.8(a), in cui però l'arco $(4, 3)$ abbia costo -5. La prima estrazione di 4 da Q non causa nessun cambiamento in $p[\cdot]$ e $d[\cdot]$, poiché $d_4 + c_{43} = 6 - 5 = 1 = d_3$. Quando però 4 viene estratto da Q la seconda volta, avendo etichetta $d_4 = 5$, ciò causa il rietichettamento di 3 in quanto $d_4 + c_{43} = 5 - 5 = 0 < 1 = d_3$. Si noti che il vettore $p[\cdot]$ non descrive più un albero; infatti si ha $p[3] = 4, p[4] = 2$ e $p[2] = 3$, cioè il ciclo $\{3, 2, 4, 3\}$. Il controllo delle condizioni di Bellman per il nodo 3 causa la diminuzione dell'etichetta di 2 al valore 1, il che causa la diminuzione dell'etichetta di 4 al valore 4, il che causa la diminuzione dell'etichetta di 3 al valore -1 e così via. In altre parole, i nodi del ciclo vengono inseriti in Q un numero infinito di volte, mentre il valore delle loro etichette diminuisce indefinitamente.

Osserviamo che *SPT* è un algoritmo di ricerca locale, in cui però ad ogni passo si lavora sulla coppia (T, d), dove T è un albero (soluzione ammissibile) e d è un'approssimazione superiore del vettore delle etichette di T. La trasformazione σ effettua una sequenza di estrazioni di nodi da Q e di controlli delle condizioni di Bellman fino a che non riesce a determinare un arco (i, j) la cui condizione di Bellman è violata, generando mediante tale arco una nuova coppia (T', d'), dove

o T' è ottenuto da T inserendo l'arco (i, j) al posto di $(p[j], j)$;

oppure $T' = T$, in quanto $p[j] = i$, e d' è una migliore approssimazione delle etichette di T rispetto a d.

L'algoritmo termina quando σ esaurisce le estrazioni dei nodi da Q ed i controlli sulle stelle uscenti senza riuscire a modificare la coppia (T, d); in questo caso, l'ottimo locale determinato risulta anche essere un ottimo globale.

L'algoritmo SPT è un algoritmo molto generale il cui effettivo comportamento dipende dal modo con cui viene implementato l'insieme Q dei nodi candidati. In effetti, ad implementazioni diverse corrispondono comportamenti molto diversi in termini di complessità computazionale. Ad alto livello possiamo pensare a due scelte alternative:

1. Q è una *coda di priorità*, cioè un insieme in cui ogni elemento ha associato un valore (chiave), e la scelta dell'elemento da estrarre avviene sulla base di questo valore;

2. Q viene implementato come una *lista* e la scelta dell'elemento da estrarre è determinata dalla posizione dell'elemento nella lista.

Tali scelte corrispondono a strategie implementative diverse, realizzabili in modi molto differenti fra loro: nel seguito discuteremo alcune di queste possibili implementazioni e le conseguenze che esse hanno sull'efficienza dell'algoritmo.

2.3.4 Algoritmi a coda di priorità

L'insieme Q viene implementato come coda di priorità; ad ogni elemento i è cioè associata una chiave di priorità, che nel nostro caso è l'etichetta d_i, e la priorità di i cresce al decrescere di d_i. Le operazioni elementari eseguibili su Q sono:

- inserimento di un elemento con l'etichetta associata,
- modifica (riduzione) dell'etichetta di un elemento di Q,
- selezione dell'elemento con etichetta minima e sua rimozione da Q.

Chiamiamo $SPT.S$ (da *Shortest-first*) la versione di SPT in cui ad ogni iterazione si estrae da Q un elemento ad etichetta minima; l'operazione "select u from Q;" viene realizzata come:

$$\text{select } u \text{ from } Q \text{ such that } d_u := \min\{d_i : i \in Q\};$$

Vale il seguente Teorema:

Teorema 2.4 [Dijkstra, 1959] *Nel funzionamento di* SPT.S *su grafi con costi non negativi, ogni nodo verrà inserito in Q (e rimosso da esso) al più una volta.*

Dimostrazione Indichiamo con u_k e $d^k[\cdot]$ rispettivamente il nodo estratto da Q ed il vettore delle etichette all'iterazione k ($u_1 = r$). Vogliamo innanzitutto dimostrare che la successione dei valori delle etichette dei nodi estratti da Q è non decrescente, ossia che $d^{k+1}[u_{k+1}] \geq d^k[u_k]$, per ogni $k \geq 1$. Per questo basta considerare due casi:

- $d^{k+1}[u_{k+1}] = d^k[u_{k+1}]$, ossia l'etichetta di u_{k+1} non è cambiata durante la k-esima iterazione. In questo caso u_{k+1} apparteneva certamente a Q all'inizio della k-esima iterazione (un nodo può entrare in Q solo se il valore della sua etichetta diminuisce) e quindi $d^{k+1}[u_{k+1}] = d^k[u_{k+1}] \geq d^k[u_k]$ perché u_k è uno dei nodi di Q con etichetta di valore minimo al momento in cui viene estratto.
- $d^{k+1}[u_{k+1}] < d^k[u_{k+1}]$, ossia l'etichetta di u_{k+1} è cambiata durante la k-esima iterazione: questo significa che u_k è il predecessore di u_{k+1} all'inizio della k+1-esima iterazione e quindi $d^{k+1}[u_{k+1}] = d^k[u_k] + c_{u_k,u_{k+1}}$, da cui, dato che $c_{u_k,u_{k+1}} \geq 0$, si ottiene ancora una volta $d^{k+1}[u_{k+1}] \geq d^k[u_k]$.

Poiché i nodi vengono inseriti in Q quando il valore della loro etichetta decresce, se un nodo entrasse in Q una seconda volta, la sua etichetta avrebbe un valore inferiore a quello che aveva nel momento della sua prima estrazione. Ciò contraddice quanto appena dimostrato. ◇

Il fatto che un nodo non possa essere inserito in Q, e quindi estratto, più di una volta fa sì che il generico arco (i, j) possa essere esaminato al più una volta, cioè quando viene selezionato il nodo i, e pertanto *SPT.S* ha complessità polinomiale. Si può invece dimostrare che, nel caso di *costi negativi*, l'algoritmo ha complessità *esponenziale* in quanto esistono grafi per i quali l'algoritmo esegue un numero esponenziale di iterazioni.

Esercizio 2.11 *Dimostrare, per* SPT.S*, che se i costi degli archi sono non negativi allora, ad ogni iterazione, il valore dell'etichetta di un nodo è il costo del cammino di T che va dall'origine a quel nodo, e non una sua approssimazione superiore.*

Sono possibili diverse implementazioni dell'algoritmo *SPT.S*, che differiscono per il modo in cui è implementata la coda di priorità Q. La scelta dell'implementazione di Q non cambia il comportamento dell'algoritmo (si può pensare che la sequenza di estrazioni da Q sia indipendente da tale scelta), ma ne influenza la complessità e l'efficienza computazionale. Nel seguito discuteremo brevemente alcune possibili implementazioni.

Lista non ordinata

Q è implementata come una lista non ordinata, ad esempio mediante un vettore a puntatori, in cui i nodi vengono inseriti in testa o in coda e la selezione del nodo di etichetta minima viene effettuata per mezzo di una scansione completa della lista. Le operazioni elementari hanno quindi le seguenti complessità:

inizializzazione delle etichette e della lista Q:	$O(n)$,
selezione del nodo di etichetta minima:	$O(n)$,
rimozione da Q del nodo di etichetta minima:	$O(1)$,
inserzione di un nodo o modifica della sua etichetta:	$O(1)$.

L'algoritmo risultante è noto come *algoritmo di Dijkstra*: è facile verificare che, su grafi con costi non negativi, questo algoritmo ha complessità $O(n^2)$. Infatti, dal Teorema (2.4) discende che non verranno estratti più di n nodi da Q: ad ogni estrazione del nodo di etichetta minima si scandisce l'intera lista in tempo $O(n)$ e quindi il costo totale delle operazioni di gestione della lista è $O(n^2)$. Siccome ogni nodo viene estratto da Q al più una volta, ogni arco (i, j) viene esaminato, una sola volta, se e quando i viene estratto da Q. Le operazioni sui singoli archi sono effettuate in tempo costante, e quindi costano complessivamente $O(m)$; dato che $m < n^2$, la complessità in tempo dell'algoritmo è $O(n^2)$.

Esempio 2.7:
Si vuole determinare l'albero dei cammini minimi di radice $r = 1$ sul grafo di figura 2.10(a) con la procedura *SPT.S*, usando per Q una lista non ordinata. L'albero fittizio iniziale è quello di figura 2.10(b).

Gli alberi da (b) a (g) sono quelli che si ottengono nell'inizializzazione e come risultato delle iterazioni dell'algoritmo riportate nella seguente tabella. Gli archi disegnati in queste figure corrispondono a quelli indicati dal vettore dei predecessori; quelli tratteggiati corrispondono agli archi fittizi di costo $M = 26$. I valori delle etichette sono indicati accanto ai nodi. Nell'ultima iterazione si seleziona il nodo 6, ma non si hanno modifiche di etichette e quindi l'albero rimane inalterato.

Iterazione	Q	u	Archi esaminati	Etichette modificate	Predecessori modificati	Albero
1	$\{1\}$	1	$(1,2),(1,3)$	d_2, d_3	p_2, p_3	(c)
2	$\{2,3\}$	3	$(3,5)$	d_5	p_5	(d)
3	$\{2,5\}$	2	$(2,3),(2,4)$	d_4	p_4	(e)
4	$\{4,5\}$	5	$(5,4),(5,6)$	d_4, d_6	p_4, p_6	(f)
5	$\{4,6\}$	4	$(4,6)$	d_6	p_6	(g)
6	$\{6\}$	6				(g)

Esercizio 2.12 *Scrivere la procedura* SPT.S *con Q implementato come lista non ordinata.*

Esercizio 2.13 *Applicare* SPT.S *con Q implementato come lista non ordinata al grafo di figura 2.9 con radice $r = 1$.*

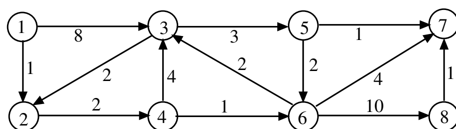

Figura 2.9:

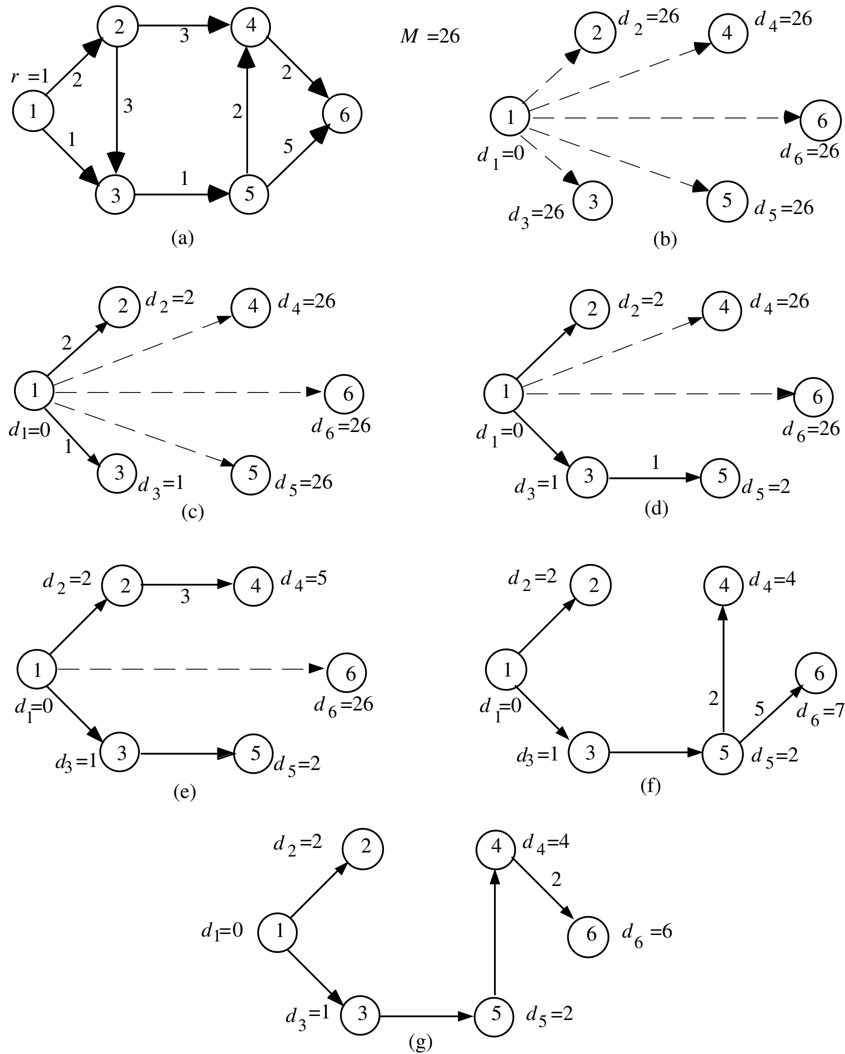

Figura 2.10: Alberi generati dall'Algoritmo *SPT.S*

Heap binario

Come abbiamo visto, la complessità dell'algoritmo di Dijkstra è fondamentalmente dovuta alla gestione dell'insieme Q: per questo, sono state proposte diverse strutture di dati per implementare Q in modo efficiente mantenendo l'insieme Q parzialmente ordinato. Una delle alternative più utilizzate è quella di realizzare Q mediante uno *heap binario*: in questo caso, il costo delle operazioni su Q è il seguente:

inizializzazione delle etichette e dello heap Q:	$O(n)$,
selezione del nodo di etichetta minima:	$O(1)$,
rimozione della radice dello heap e suo riordinamento:	$O(\log n)$,
inserzione di un nodo o modifica della sua etichetta:	$O(\log n)$.

Se i costi degli archi sono non negativi, le operazioni di ordinamento dello heap a seguito di inserimenti o rimozioni di nodi da Q sono al più $m + n$: pertanto, la versione di *SPT.S* che utilizza un heap binario ha complessità $O(m \log n)$. Si noti che tale complessità è migliore di quella dell'algoritmo di Dijkstra nel caso di grafi *sparsi* ($m \approx n$), mentre è peggiore di quella dell'algoritmo di Dijkstra nel caso di grafi *densi* ($m \approx n^2$).

Sono stati proposte molte implementazioni di Q basate su differenti implementazioni di code di priorità, quali ad esempio i *Buckets*, i *Radix Heaps* ed i *Fibonacci Heaps*; per ulteriori dettagli si rinvia alla letteratura citata.

2.3.5 Algoritmi a selezione su lista

In questi algoritmi l'insieme Q viene implementato come una *lista*, cioè una *sequenza* di elementi su cui possono essere effettuate operazioni di rimozione ed inserzione alle estremità della sequenza, chiamate rispettivamente *testa* e *coda* della lista. Le operazioni elementari eseguibili sulle liste sono:

- inserimento di un elemento come nuova testa della lista,
- inserimento di un elemento come nuova coda della lista,
- rimozione dell'elemento testa dalla lista.

Si noti che l'aggiornamento dell'etichetta di un nodo che appartiene a Q non influisce sulla posizione dell'elemento nella lista, ossia non causa la rimozione del nodo da Q ed il suo reinserimento in una posizione diversa (formalmente, questo corrisponde al controllo **if** $v \notin Q \ldots$ nell'algoritmo *SPT*).

Si distinguono diversi tipi di liste, tra cui, nel nostro caso, hanno particolare rilevanza le seguenti:

***fila*:** l'inserzione viene effettuata in coda e la rimozione dalla testa (regola *FIFO*);

***pila*:** l'inserzione e la rimozione vengono effettuate in testa (regola *LIFO*);

***deque*:** (double-ended queue) o lista a doppio ingresso: l'inserzione viene effettuata sia in testa sia in coda e la rimozione solo dalla testa.

Indichiamo nel seguito con *SPT.L* le versioni di *SPT* nelle quali l'insieme Q è implementato come lista. La lista può essere realizzata in diversi modi (lista a puntatori, vettore di puntatori, lineare o circolare, semplice o doppia, ecc.), ed è sempre possibile fare in modo che le operazioni elementari ed il controllo di appartenenza di un elemento alla lista abbiano complessità costante, $O(1)$. La complessità di *SPT.L*, anche nel caso di costi negativi, dipende quindi linearmente dal numero di controlli delle condizioni di Bellman sugli archi uscenti dai nodi estratti da Q.

Fila

Esaminiamo ora l'algoritmo che si ottiene realizzando la lista Q come *fila* (*queue*), conosciuto in letteratura come *algoritmo di Bellman*: l'inserzione dei nodi avviene in coda e la rimozione dalla testa (regola FIFO).

Esercizio 2.14 *Scrivere la procedura* SPT.L *in cui Q è una fila.*

Esempio 2.8:
Si vuole determinare l'albero dei cammini minimi di radice $r = 1$ sul grafo in figura 2.10(a) con $SPT.L$ e Q implementata come una *fila*. Gli alberi che vengono man mano costruiti sono indicati in figura 2.11. L'albero fittizio iniziale è in (a). La simbologia nelle figure e nella tabella seguente coincide con quella utilizzata nell'esempio 2.7.

Itera-zione	Q	u	Archi esaminati	Etichette modificate	Predecessori modificati	Albero
1	$\{1\}$	1	$(1,2), (1,3)$	d_2, d_3	p_2, p_3	(b)
2	$\{2,3\}$	2	$(2,3), (2,4)$	d_4	p_4	(c)
3	$\{3,4\}$	3	$(3,5)$	d_5	p_5	(d)
4	$\{4,5\}$	4	$(4,6)$	d_6	p_6	(e)
5	$\{5,6\}$	5	$(5,4), (5,6)$	d_4	p_4	(f)
6	$\{6,4\}$	6				(f)
7	$\{4\}$	4	$(4,6)$	d_6	p_6	(g)
8	$\{6\}$	6				(g)

Esercizio 2.15 *Applicare* SPT.L*, con Q implementata come fila, al grafo di figura 2.9 con radice $r = 1$.*

L'utilizzo di una strategia FIFO corrisponde ad una "visita a ventaglio" (bfs) del grafo; si può dimostrare che tale strategia garantisce che, in assenza di cicli negativi, nessun nodo sia inserito più di $n-1$ volte in Q, e quindi che il numero di volte che si esamina un nodo o un arco è limitato superiormente da n. Siccome tutte le operazioni sui nodi (estrazione e rimozione da Q) e sugli archi (loro scansione, controllo della condizione di Bellman, eventuale aggiornamento di predecessore ed etichetta) sono implementabili in modo da avere complessità costante, la complessità della procedura è $O(mn)$, ed è data dal massimo numero di volte che si esamina lo stesso arco $(n-1)$ per il numero di archi (m).

$SPT.L$ con Q implementato come una fila può essere utilizzata per controllare se un grafo orientato possiede cicli negativi. Infatti poiché, in assenza di cicli negativi, un nodo non può essere estratto da Q più di $n-1$ volte, è sufficiente contare il numero di estrazioni da Q di ciascun nodo: appena un nodo viene estratto per l'n-esima volta, si può affermare che quel nodo appartiene ad un ciclo negativo. Il ciclo può poi essere percorso all'indietro, a partire dal nodo trovato, per mezzo del predecessore $p[\cdot]$.

Liste a doppio ingresso

Nella letteratura scientifica sono state proposte diverse realizzazioni dell'insieme Q come lista. Molto utilizzata è la lista a doppio ingresso, o *deque*,

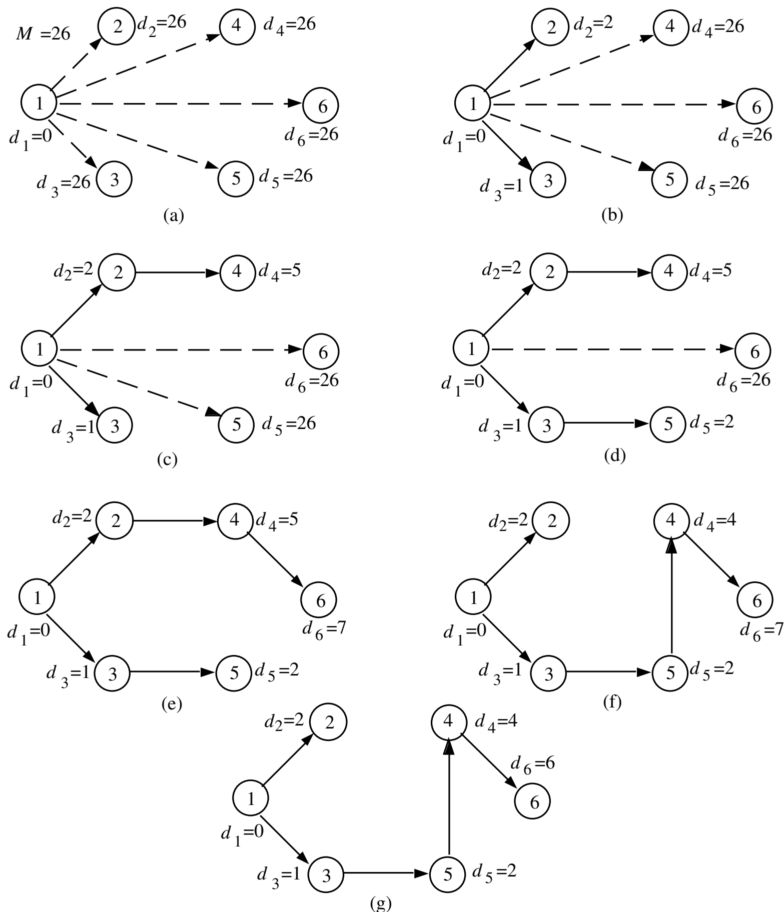

Figura 2.11: Alberi generati da $SPT.L$ con Q implementata con una fila

in cui i nodi sono inseriti in coda a Q la prima volta, mentre tutte le altre volte vengono inseriti in testa a Q. Si ottiene pertanto una politica ibrida $LIFO-FIFO$; in effetti, la lista Q può essere interpretata come una coppia di liste Q' e Q'' connesse in serie, vedi figura 2.12. Q' conterrà solo i nodi reinseriti in Q mentre Q'' conterrà solo i nodi inseriti per la prima volta in Q ed ancora non rimossi. Il nodo testa di Q'' viene rimosso solo se $Q' = \emptyset$, pertanto Q' è una *pila* (stack) e Q'' è una *fila* (queue).

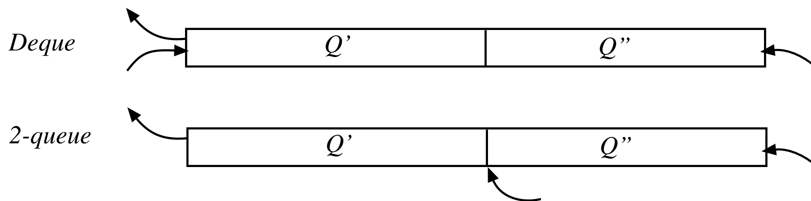

Figura 2.12: Liste a doppio ingresso

Esercizio 2.16 *Applicare SPT.L con l'uso della* deque *al grafo di figura 2.9 con radice* $r = 1$.

La motivazione per l'uso di una deque risiede nel fatto che, se un nodo i viene inserito in Q dopo essere stato precedentemente rimosso, la sua etichetta d_i è stata utilizzata per aggiornare etichette di altri nodi, discendenti di i nell'albero corrente. Una nuova inserzione di i in Q avviene poiché d_i è stata diminuita; appare pertanto conveniente correggere quanto prima le etichette dei discendenti di i (sia pure senza ricalcolare immediatamente tutte le etichette del sottoalbero), in modo da evitare il più possibile che vengano compiute iterazioni con valori delle etichette che rappresentano una "cattiva" approssimazione del valore reale del costo del cammino.

La versione di *SPT.L* in cui Q è implementata come *deque* ha però complessità esponenziale $O(n2^n)$; esistono infatti grafi per i quali una tale versione effettua un numero esponenziale di inserimenti e rimozioni di nodi da Q. Comunque, l'analisi della complessità computazionale nel caso peggiore fornisce solo una misura di salvaguardia nella crescita del numero di operazioni: infatti, nei problemi reali in cui le reti sono abbastanza *sparse* ($m \approx n$) questa variante ha un comportamento molto buono, anzi spesso risulta il più efficiente algoritmo per i cammini minimi. Ciò è particolarmente vero per reti stradali, in cui si è osservato sperimentalmente che il numero medio di estrazioni di uno stesso nodo da Q è inferiore a 2.

Un'alternativa all'uso della deque consiste nell'implementare anche Q' come fila; per questo basta mantenere un puntatore all'ultimo elemento della porzione Q' di Q ed effettuare gli inserimenti successivi al primo nella coda di Q' (vedi figura 2.12). I nodi verranno così inseriti, la prima volta, in coda a Q'' (che coincide con la coda di Q), le altre volte in coda a Q'. La struttura è conosciuta come *doppia coda* (o 2-queue).

L'algoritmo risultante risulta sperimentalmente molto efficiente e, dal punto di vista teorico, ha complessità polinomiale: si può infatti dimostrare che il massimo numero di inserimenti dello stesso nodo in Q' è $O(n^2)$, e pertanto la complessità dell'algoritmo è $O(mn^2)$.

Sono stati proposte altre implementazioni di Q basate su idee analoghe, ad esempio introducendo una "soglia" (*threshold*) opportunamente calcolata per decidere, in base al valore dell'etichetta d_i, se il nodo i sia da inserire in Q' oppure in Q''; per ulteriori dettagli si rinvia alla letteratura citata.

2.3.6 Cammini minimi su grafi aciclici

Un grafo orientato è detto *aciclico* se non contiene cicli orientati. È immediato verificare che un grafo orientato è aciclico se e solo se è possibile numerare i suoi nodi in modo tale che:

$$(i, j) \in A \Rightarrow i < j. \tag{2.7}$$

Il problema di verificare se un dato grafo orientato è aciclico e, in caso affermativo, di numerare i nodi del grafo in modo da soddisfare la proprietà (2.7), può essere risolto per mezzo di una visita del grafo (si veda il paragrafo 2.2) ed ha pertanto complessità $O(m)$.

Esercizio 2.17 *Scrivere una procedura che, in $O(m)$, controlli se un grafo orientato è aciclico e, in caso positivo, ne numeri i nodi in modo che sia soddisfatta la proprietà (2.7) (suggerimento: se un grafo è aciclico, deve esistere almeno un nodo con stella entrante vuota; eliminando tale nodo e gli archi uscenti da esso, il sottografo indotto risulta a sua volta aciclico).*

Nel seguito è descritta una procedura per il problema della determinazione dell'albero dei cammini minimi, di radice 1, su un grafo aciclico i cui nodi sono stati numerati in accordo alla (2.7); si lascia per esercizio la dimostrazione della sua correttezza e del fatto che la sua complessità è $O(m)$.

```
Procedure SPT.Acyclic (p, d):
  begin
    p[1] := nil; d[1] := 0;
    for i := 2 to n do begin p[i] := 1; d[i] := M end;
    for i := 1 to n − 1 do
      for each (i, j) ∈ FS(i) do
        if d[i] + c[i, j] < d[j] then
          begin d[j] := d[i] + c[i, j]; p[j] := i end
  end.
```

Procedura 2.3: Algoritmo *SPT.Acyclic*

Esercizio 2.18 *Applicare* SPT.Acyclic *al grafo di figura 2.9, a cui sono stati eliminati gli archi* $(3, 2)$ *e* $(6, 3)$, *con radice* $r = 1$, *dopo aver eventualmente rinumerato i nodi.*

2.3.7 Cammini minimi con radici multiple

In alcuni casi è necessario risolvere il seguente problema: dato un grafo $G = (N, A)$ con costi sugli archi ed un insieme non vuoto R di nodi "radice", determinare per ogni nodo $i \notin R$ il cammino minimo da uno dei nodi $r \in R$ ad i. In altre parole, si vuole determinare, per ogni nodo i, la "radice" dalla quale sia più conveniente raggiungerlo, ossia la radice alla quale corrisponde il cammino meno costoso fino ad i.

È facile verificare che questo problema può essere risolto in modo analogo a quanto visto per la procedura di visita nel paragrafo 2.2.2, ossia mediante un'applicazione della procedura *SPT* al grafo $G' = (N', A')$ in cui $N' = N \cup \{s\}$, dove s è un nodo fittizio che svolge il ruolo di "super-radice", $A' = A \cup \{(s, r) : r \in R\}$ ed i costi degli archi in $A' \setminus A$ (uscenti da s) sono

nulli. L'albero dei cammini minimi su G' fornisce una soluzione ottima al problema dei cammini minimi con radici multiple.

Esempio 2.9:
Si vogliono determinare i cammini minimi rispetto all'insieme di radici $R = \{1, 5\}$ sul grafo di figura 2.9. In figura 2.13 è mostrato il corrispondente grafo G' ed il relativo albero dei cammini minimi (archi in grassetto). Quindi, per i nodi 2 e 4 è conveniente selezionare il nodo 1 come radice, mentre per i nodi 3, 6, 7 e 8 è conveniente selezionare il nodo 5 come radice.

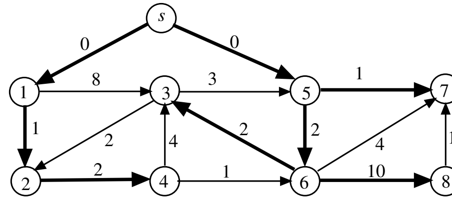

Figura 2.13:

Analogamente a quanto visto per la procedura di visita, è possibile modificare la procedura *SPT* in modo che lavori direttamente sul grafo originale G, senza la necessità di costruire fisicamente G'; è solamente necessario sostituire le istruzioni

$$p[r] := nil; \quad d[r] := 0; \quad Q := \{r\};$$

con

$$\textbf{for each } r \in R \textbf{ do begin } p[r] := nil;\ d[r] := 0 \textbf{ end}; \quad Q := R;$$

Esercizio 2.19 *Dato che il grafo G' è sicuramente aciclico se il grafo originale G lo è (s può essere numerato con un indice minore di tutti quelli dei nodi originali, ad esempio 0), si mostri che è possibile risolvere il problema dei cammini minimi con radici multiple su un grafo aciclico in $O(m)$ modificando opportunamente la procedura* SPT.Acyclic.

2.4 Albero di copertura di costo minimo

Riprendiamo il problema dell'*albero di copertura di costo minimo* (MST), già presentato nel paragrafo 1.2.2.2. Dato un grafo non orientato $G = (N, A)$ con costi c_{ij} associati agli archi, consideriamo il problema della determinazione di un albero di copertura $T = (N, A_T)$ tale che sia minimo il costo di T, definito come $c(T) = \sum_{(i,j)\in A_T} c_{ij}$. Senza alcuna perdita di generalità possiamo assumere che G sia connesso e che i costi siano positivi: la connessione può infatti essere ottenuta con l'aggiunta di archi a costo opportunamente grande ($M = c_{max} + 1$), mentre la positività dei costi può essere ottenuta sommando al costo di ogni arco una opportuna costante C (di conseguenza, il costo di ogni soluzione viene incrementato del valore $(n - 1)C$).

Il problema (MST) può essere risolto per mezzo di un algoritmo di tipo *greedy*. Presenteremo qui uno schema algoritmico molto generale, *Greedy-MST*, per la sua risoluzione.

Tale algoritmo costruisce l'albero incrementalmente, mantenendo ad ogni passo due insiemi di archi: S, l'insieme degli archi già inseriti nell'albero, e R, l'insieme degli archi scartati, cioè l'insieme degli archi che certamente non verranno inseriti nell'albero. S e R, che all'inizio sono vuoti, vengono aggiornati per mezzo delle seguenti operazioni:

Inserzione: Seleziona in G un taglio (N', N'') tale che $S \cap A(N', N'') = \emptyset$, ed aggiungi ad S un arco (u, v) per cui $c_{uv} = \min\{c_{ij} : (i, j) \in A(N', N'') \setminus R\}$.

Cancellazione: Seleziona in G un ciclo C tale che $C \cap R = \emptyset$, e aggiungi a R un arco $(u, v) \in C \setminus S$ per cui $c_{uv} = \max\{c_{ij} : (i, j) \in C \setminus S\}$.

Procedure *Greedy-MST* (G, c, A_T):
    begin
        $S := \emptyset$; $R := \emptyset$;
        repeat Applica *Inserzione* o *Cancellazione*
        until $S \cup R = A$ **or** $|S| = n - 1$;
        $A_T := S$
    end.

Procedura 2.4: Algoritmo *greedy* per il problema (MST)

L'algoritmo termina quando nessuna delle due operazioni è più applicabile, cioè quando risulta $S \cup R = A$, oppure quando sono stati inseriti $n - 1$ archi in S, cioè quando il grafo parziale definito da S è un albero. Quando l'algoritmo termina, S definisce una soluzione ottima del problema; vale infatti la seguente proprietà:

Lemma 2.2 *Se la coppia di insiemi (S, R) è tale per cui esiste in G un albero di copertura di costo minimo $T = (N, A_T)$ con $S \subseteq A_T$ e $R \cap A_T = \emptyset$, allora l'applicazione di una qualsiasi delle operazioni di* inserzione *o di* cancellazione *produce una nuova coppia (S, R) che gode della stessa proprietà.*

Dimostrazione Cominciamo col dimostrare che la tesi vale per l'operazione di *inserzione*. Sia T un albero ottimo tale che A_T contiene S e $R \cap A_T = \emptyset$; siano (N', N'') e (u, v), rispettivamente, il taglio e l'arco selezionati dall'operazione di *inserzione*. Se $(u, v) \in A_T$ allora la proprietà è soddisfatta. Altrimenti, essendo T connesso, esiste almeno un arco (k, l) dell'unico cammino che connette u e v in T che appartenga ad $A(N', N'')$. Sostituendo (u, v) a (k, l) in A_T, si ottiene un nuovo albero di copertura T'; essendo $c_{uv} \leq c_{kl}$, si ha $c(T') \leq c(T)$. Siccome per ipotesi T è ottimo, anche T' lo è (e quindi, $c_{uv} = c_{kl}$); T' contiene l'arco selezionato (u, v), e quindi la proprietà è soddisfatta da T' (anche se non da T).

Consideriamo ora l'operazione di *cancellazione*. Sia T un albero ottimo per cui $A_T \cap R = \emptyset$, e siano C e (u, v), rispettivamente, il ciclo e l'arco selezionati dall'operazione di *cancellazione*. Se T non contiene (u, v), allora la proprietà è soddisfatta. Altrimenti, cancellando (u, v) dall'albero si ottengono due sottoalberi $T' = (N', A'_T)$ e $T'' = (N'', A''_T)$; tra gli archi del ciclo C deve necessariamente esistere un arco $(k, l) \notin A_T$ che appartenga a $A(N', N'')$, con $c_{kl} \leq c_{uv}$. Sostituendo (u, v) con (k, l) in T si ottiene un nuovo albero, anch'esso ottimo, che soddisfa la proprietà (si ha nuovamente $c_{kl} = c_{uv}$). $\diamond$

Teorema 2.5 *L'algoritmo* Greedy-MST *termina fornendo un albero di copertura di costo minimo.*

Dimostrazione La prova è per induzione: basta osservare che all'inizio S e R sono vuoti, e quindi godono banalmente della proprietà del precedente lemma, proprietà che si mantiene valida ad ogni iterazione successiva. L'algoritmo termina in al più m iterazioni, perché ad ogni iterazione la cardinalità dell'insieme $S \cup R$ aumenta di una unità. $\diamond$

Si noti che la correttezza dell'algoritmo non dipende dall'ordine con cui vengono realizzate le operazioni di *inserzione* e *cancellazione*. In effetti, esistono diverse possibili implementazioni dell'algoritmo *Greedy-MST* che si distinguono per l'ordine con cui vengono effettuate tali operazioni: nel seguito presenteremo due di tali algoritmi, l'algoritmo di Kruskal e l'algoritmo di Prim.

2.4.1 Algoritmo di Kruskal

In questo algoritmo, gli archi del grafo vengono inizialmente ordinati in ordine di costo non decrescente. Seguendo tale ordinamento, ad ogni iterazione viene selezionato il primo arco (u, v) non ancora esaminato: se (u, v) non forma alcun ciclo con gli archi in S, allora esso viene inserito in S, cioè si applica l'operazione di *inserzione*, altrimenti l'arco viene inserito in R, cioè si applica l'operazione di *cancellazione*. Le due operazioni sono applicabili legittimamente. Infatti, nel primo caso, la non esistenza di cicli nel grafo parziale (N, S) garantisce l'esistenza di un taglio (N', N'') con $u \in N'$ e $v \in N''$, tale che $\{(i, j) : i \in N', j \in N''\} \cap S = \emptyset$; inoltre, poiché gli archi non ancora selezionati hanno un costo non minore di c_{uv}, è vero che (u, v) è un arco di costo minimo fra gli archi del taglio. Nel secondo caso, tutti gli archi del ciclo C appartengono ad S tranne l'arco (u, v): quindi (i, j) è l'arco di costo massimo tra quelli in $C \setminus S$, essendo l'unico. Si noti comunque che (u, v), essendo stato selezionato dopo tutti gli altri archi di C, è anche l'arco di costo massimo fra tutti quelli del ciclo.

Una descrizione dettagliata dell'algoritmo è la seguente, dove la funzione $Sort(A)$ restituisce l'insieme A ordinato per costo non decrescente, e la funzione $Component(S, (u, v))$ risponde *true* se u e v appartengono alla stessa componente connessa del grafo parziale definito da S, cioè se (u, v) induce un ciclo su tale grafo, e *false* altrimenti.

L'operazione più critica dell'algoritmo è il controllo di esistenza di un ciclo comprendente l'arco (u, v), cioè se due nodi u e v appartengono alla stessa componente connessa o meno. È possibile predisporre delle opportune strutture di dati che permettano di effettuare in modo efficiente questo controllo. Con l'ausilio di tali strutture di dati, e considerando che il costo dell'ordinamento degli archi è $O(m \log n)$, si può ottenere una complessità computazionale pari a $O(m \log n)$.

Procedure *Kruskal* (G, c, A_T):
  begin
    $S := \emptyset$; $R := \emptyset$; $X := Sort(A)$;
    repeat
      Estrai da X il primo arco (u, v);
      if $Component(S, (u, v))$
        then $R := R \cup \{(u, v)\}$ (cancellazione)
        else $S := S \cup \{(u, v)\}$ (inserzione)
    until $|S| = n - 1$;
    $A_T := S$
  end.

Procedura 2.5: Algoritmo di Kruskal per il problema (MST)

Esempio 2.10:
Consideriamo il grafo in figura 2.14 e applichiamo ad esso l'algoritmo di Kruskal; in figura 2.15 sono riportati i passi effettuati, riportando gli archi inseriti in S ed il costo $c(S)$. L'ordinamento iniziale fornisce $X = \{(2, 4), (3, 4), (5, 7), (1, 3), (1, 2), (2, 3), (4, 5), (4, 7), (3, 6), (2, 5), (6, 7), (4, 6)\}$.

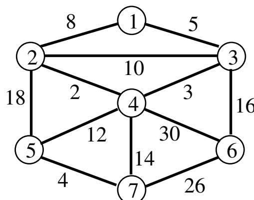

Figura 2.14: un'istanza del problema (MST)

Si noti che il quinto arco inserito, $(4, 5)$, ha costo $c_{45} = 12$; tale inserimento avviene dopo la cancellazione degli archi $(1, 2)$ e $(2, 3)$ in quanto $(1, 2)$ forma ciclo con $(2, 4)$, $(3, 4)$ e $(1, 3)$, mentre $(2, 3)$ forma ciclo con $(2, 4)$ e $(3, 4)$. Analogamente, prima dell'inserzione $(3, 6)$ vi è stata la cancellazione dell'arco $(4, 7)$ in quanto forma ciclo con $(5, 7)$ e $(4, 5)$. Con l'inserzione di $(3, 6)$ il grafo parziale definito da S diviene connesso ($|S| = 6$), e quindi un albero di copertura di costo minimo; gli ultimi tre archi nell'insieme $X = \{(2, 5), (6, 7), (4, 6)\}$ non vengono esaminati.

Esercizio 2.20 *Applicare* Kruskal *al grafo di figura 2.16; fornire ad ogni iterazione la foresta $T = (N, S)$ e l'arco esaminato indicando se viene eliminato o inserito.*

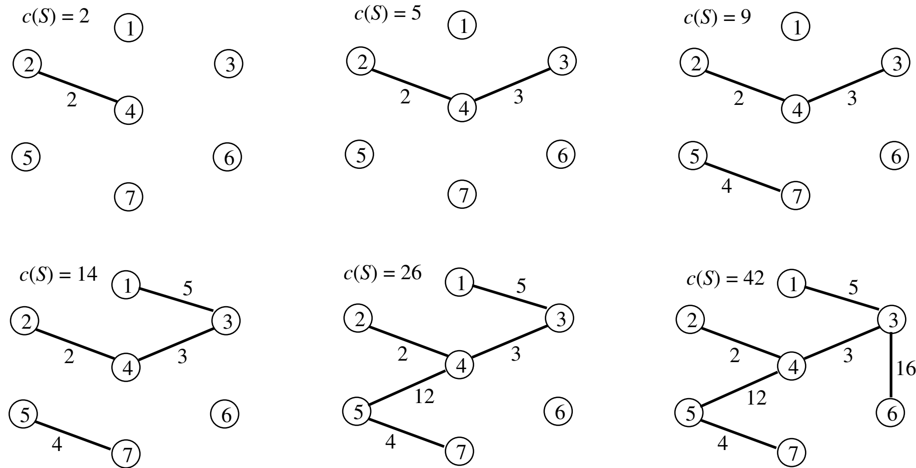

Figura 2.15: Passi effettuati dall'algoritmo di Kruskal

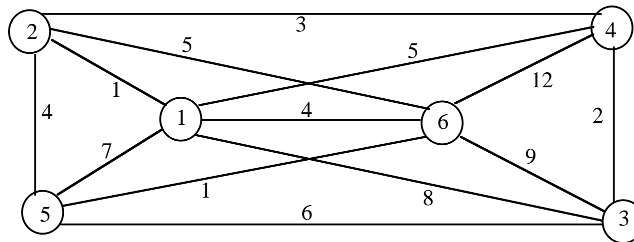

Figura 2.16:

Esercizio 2.21 *Se il grafo G non è connesso, non esiste un albero di copertura per G; esiste però una foresta di alberi di copertura, e quindi alberi di copertura di costo minimo, per ciascuna delle componenti connesse. Si discuta come modificare l'algoritmo di Kruskal, senza aumentarne la complessità, in modo tale che determini una foresta di alberi di copertura di costo minimo per tutte le componenti connesse di G.*

2.4.2 Algoritmo di Prim

L'*algoritmo di Prim* effettua solamente l'operazione di *inserzione*, mentre l'operazione di *cancellazione* viene effettuata implicitamente. Per effettuare l'operazione di *inserzione*, viene costruito ed aggiornato ad ogni iterazione un taglio (N', N'') con la caratteristica che (N', S) è un albero di copertura di costo minimo per il grafo indotto da N'. Ad ogni iterazione viene selezionato ed inserito in S un arco a costo minimo fra quelli che hanno un estremo in N' e l'altro in N'', cioè un arco appartenente all'insieme $A(N', N'')$ degli archi del taglio.

L'insieme N' viene inizializzato con un nodo arbitrario r: $N' = \{r\}$ e $N'' = N \setminus \{r\}$; pertanto $S = \emptyset$. Analogamente a quanto fatto per i problemi

di cammino minimo, introduciamo un arco fittizio (r, i), per ogni $i \in N \setminus \{r\}$, di costo opportunamente grande $c_{ri} = M$. Se G è connesso, dopo l'inserimento di $n-1$ archi in S, l'insieme N' coincide con N e quindi $T = (N, S)$ è un albero di copertura di costo minimo per G. Altrimenti si otterrà un taglio (N', N'') i cui archi sono tutti fittizi; in tal caso l'algoritmo si arresta connettendo questi nodi ad r mediante gli archi fittizi. Per memorizzare la porzione di albero corrente definita da (N', S), e alla fine l'albero di copertura T, utilizziamo un vettore di predecessori $p[\cdot]$.

Per effettuare l'operazione di inserzione si deve determinare un arco di costo minimo appartenente all'insieme corrente $A(N', N'')$. Al fine di rendere efficiente questa operazione, che in generale avrebbe complessità in tempo $O(m)$, memorizziamo, per ogni nodo $j \in N''$, sia l'arco $(i, j) \in S(i) \cap A(N', N'')$ di costo minimo, utilizzando il vettore di predecessori $(p[j] = i)$ che il suo costo, utilizzando un vettore di etichette $d[\cdot]$ $(d[j] = c_{ij})$. In tal modo è possibile determinare, in tempo $O(n)$, l'arco di costo minimo in $A(N', N'')$ selezionando un nodo di etichetta minima tra i nodi in N'':

$$u \in N'' \ : \ \ d_u = \min\{d_j \ : \ j \in N''\}.$$

L'arco che viene inserito in S è quindi $(p[u], u)$ e il suo costo è $c_{p[u]u} = d_u$.

Lo spostamento di u da N'' a N' equivale all'inserimento di $(p[u], u)$ in S, e alla modifica del taglio ($N' = N' \cup \{u\}$ e $N'' = N'' \setminus \{u\}$) e dell'insieme degli archi del taglio. Si noti che, per aggiornare i valori $p[j]$ e $d[j]$ per ogni nodo $j \in N''$, è sufficiente esaminare ciascun arco $(u, v) \in S(u)$ tale che $v \in N''$ e verificare se esso non sia l'arco del taglio incidente in v di costo minimo. Infatti, basta scegliere l'"arco di costo minore" tra (u, v) e $(p[v], v)$, cioè basta controllare se $c_{uv} < d_v$. In caso affermativo (u, v) risulta migliore di $(p[v], v)$ e viene sostituito ad esso ponendo $p[v] = u$ e $d_v = c_{uv}$; altrimenti non si effettua nessun cambiamento.

Si noti che, nel primo caso, si ha un'operazione di cancellazione implicita dell'arco $(p[v], v)$, mentre nel secondo viene cancellato l'arco (u, v).

Per ragioni implementative, viene utilizzato un insieme Q dei nodi candidati che contenga tutti e soli i nodi di N'' per i quali esista almeno un arco del taglio incidente in essi; in tal modo la ricerca del nodo di etichetta minima viene effettuato in Q. Si noti che i nodi $j \in N'' \setminus Q$ sono i nodi con etichetta arbitrariamente grande $(d[j] = M)$.

L'algoritmo è descritto in modo da evidenziarne la forte somiglianza con l'algoritmo *SPT.S*. In esso viene modificata la condizione di scelta dell'arco: al posto della condizione di Bellman si inserisce la condizione di "arco di costo minore". Inoltre, per indicare che un nodo i appartiene all'insieme N', e quindi che non devono essere più effettuate inserzioni di archi incidenti in esso, si pone $d[j] = -M$.

È facile verificare che, nel caso in cui Q sia implementata come una lista non ordinata, la complessità dell'algoritmo di Prim è $O(n^2)$, come quella

del suo corrispondente per il problema dei cammini minimi. Infatti, anche in questo caso non si avranno più di n estrazioni di nodi da Q, e ogni estrazione ha costo $O(n)$ per la scansione di Q per determinare il nodo di etichetta minima. Le operazioni relative agli archi hanno complessità costante e saranno ripetute al più due volte per ciascun arco: quindi, globalmente costano $O(m) = O(n^2)$ poiché $m < n^2$. Se Q è implementato come uno *Heap* la complessità è $O(m \log n)$ (si veda il paragrafo 2.3.4).

```
Procedure Prim (G, c, r, p):
  begin
    for i := 1 to n do begin p[i] := r; d[i] := M end;
    p[r] := nil; d[r] := −M; Q := {r};
    repeat
      seleziona u in Q tale che d[u] = min{d[j] : j ∈ Q};
      d[u] := −M; Q := Q \ {u};
      for each (u, v) ∈ S(u) do
        if c[u, v] < d[v] then
          begin d[v] := c[u, v]; p[v] := u;
            if v ∉ Q then Q := Q ∪ {v} end
    until Q = ∅
  end.
```

Procedura 2.6: Algoritmo *Prim*

Esempio 2.11:
In figura 2.17 sono rappresentate le soluzioni al termine delle iterazioni dell'algoritmo di Prim applicato al grafo di figura 2.14. Gli archi evidenziati sono quelli inseriti in S, e sono incidenti in nodi i tali che $p[i]$ è fissato e $d[i] = -M = -31$, mentre gli archi tratteggiati sono gli archi $(p[v], v)$ candidati ad essere inseriti in S, per ogni $v \in Q$ (accanto al nodo è riportata la sua etichetta). La linea tratteggiata indica il taglio (N', N''). I nodi evidenziati sono quelli di etichetta minima che verranno estratti da Q all'iterazione successiva. La sequenza dei valori delle etichette dei nodi rimossi da Q è $0, 5, 3, 2, 12, 4, 16$. Ad ogni iterazione viene riportato il costo $c(S)$ della soluzione corrente; al termine $c(S) = 42$, cioè la somma delle etichette dei nodi al momento della loro estrazione da Q; infatti, esse rappresentano i costi degli archi che formano l'albero stesso.

Esercizio 2.22 *Applicare* Prim *al grafo di figura 2.16; fornire ad ogni iterazione l'albero parziale $T = (N', A_T)$, l'insieme Q e il nodo u selezionato.*

Esercizio 2.23 *Si discuta come modificare l'algoritmo di Prim, senza aumentarne la complessità, in modo tale che, anche se applicato ad un grafo G non è connesso, determini alberi di copertura di costo minimo per tutte le componenti connesse di G.*

2.4.3 Albero di copertura bottleneck

Un diverso problema di albero ottimo, strettamente correlato con (MST), è quello in cui si richiede di minimizzare non il costo dell'albero di copertura

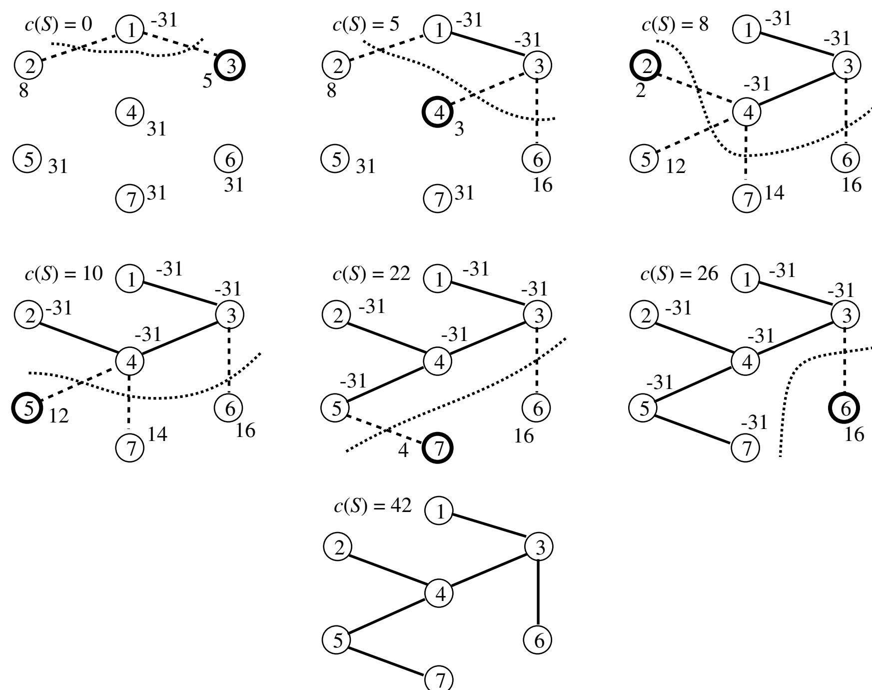

Figura 2.17: Passi effettuati dall'algoritmo di Prim

$T = (N, A_T)$ ma il suo *valore bottleneck (collo di bottiglia)* $V(T) = \max\{c_{ij} : (i,j) \in A_T\}$, ossia il massimo costo degli archi utilizzati.

Vale il seguente lemma.

Lemma 2.3 *Un albero di copertura di costo minimo è anche un albero bottleneck.*

Dimostrazione Sia $T' = (N, A_{T'})$ un albero di copertura di costo minimo e sia (u,v) un suo arco di costo massimo, cioè $c_{uv} = \max\{c_{ij} : (i,j) \in A_{T'}\}$. La rimozione di (u,v) da T' genera un taglio (N', N'') del grafo; per l'ottimalità di T' non esistono archi del taglio di costo inferiore a c_{uv} e quindi il valore $V(T)$ di qualsiasi albero bottleneck T è tale che $V(T) \geq c_{uv}$. Da $V(T') = c_{uv}$ segue che T' è un albero bottleneck. $\diamond$

Si noti che, in generale, non è vero il viceversa; infatti, dato un albero di copertura di costo minimo (e quindi anche bottleneck) T', di valore bottleneck $V(T')$, se si sostituisce un qualsiasi arco (i,j) con un arco (p,q) appartenente agli archi del taglio indotto da (i,j), purché $c_{ij} < c_{pq} \leq V(T')$, si ottiene un nuovo albero T, che è bottleneck ma non più di costo minimo.

Abbiamo pertanto mostrato che i due problemi non sono equivalenti, ma che l'insieme delle soluzioni del problema dell'albero bottleneck contiene l'insieme delle soluzioni del problema dell'albero di copertura di costo minimo; pertanto per risolvere il primo problema è sufficiente risolvere il secondo.

2.5 Il problema di flusso massimo

Sia $G = (N, A)$ un grafo orientato su cui sia definito un vettore $u = [u_{ij}]$ di capacità superiori associate agli archi; inoltre, siano s e t due nodi distinti, detti rispettivamente *origine* (o *sorgente*) e *destinazione* (o *pozzo*). Il *problema di flusso massimo* consiste nel determinare la massima quantità di flusso che è possibile inviare da s a t attraverso G; più precisamente, si vuole determinare il massimo valore v per cui ponendo $b_s = -v$, $b_t = v$ e $b_i = 0$ per ogni $i \neq s, t$, esiste un flusso ammissibile x.

Una formulazione analitica del problema è quindi la seguente:

$$
\begin{array}{lllllll}
\max & & & & v & & \\
& \sum_{(j,s)\in BS(s)} x_{js} & - & \sum_{(s,j)\in FS(s)} x_{sj} & +v & = 0 & \\
& \sum_{(j,i)\in BS(i)} x_{ji} & - & \sum_{(i,j)\in FS(i)} x_{ij} & & = 0 & i \in N \setminus \{s,t\} \\
& \sum_{(j,t)\in BS(t)} x_{jt} & - & \sum_{(t,j)\in FS(t)} x_{tj} & -v & = 0 & \\
& & & 0 \leq x_{ij} \leq u_{ij} & & & (i,j) \in A
\end{array}
\tag{2.8}
$$

Data una soluzione ammissibile (x, v), il valore v, che determina il bilancio ai nodi s e t, è detto *valore del flusso* x.

Il problema di flusso massimo è in realtà un caso particolare del problema di flusso di costo minimo. Infatti, la formulazione (2.8) può essere vista come quella del problema di (MCF) su un grafo G' ottenuto da G aggiungendo un arco fittizio (t, s), detto *arco di ritorno*, il cui flusso x_{ts} è proprio il valore v: la colonna dei coefficienti relativa alla variabile v, interpretata come una colonna della matrice di incidenza di G', individua proprio l'arco (t, s), come mostrato anche nella figura 2.18.

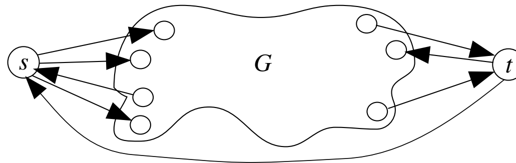

Figura 2.18: Un grafo con l'arco di ritorno (t, s)

In G' i nodi sono di trasferimento, compresi s e t, cioè $b = 0$; un tale problema di flusso è detto *di circolazione*. I costi degli archi sono nulli salvo quello dell'arco (t, s) che è posto uguale a -1: di conseguenza, minimizzare $-v$ equivale a massimizzare il valore v del flusso che transita lungo l'arco (t, s), ossia del flusso che in G è inviato da s a t.

Esempio 2.12:
Si consideri il grafo in figura 2.19, in cui la sorgente è $s = 1$ ed il pozzo è $t = 6$.

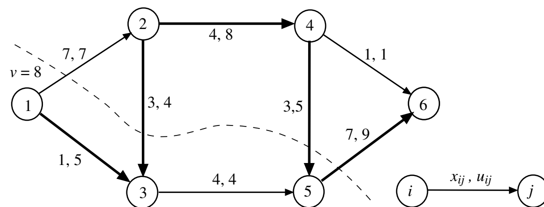

Figura 2.19: un grafo orientato con un flusso x ammissibile di valore $v = 8$

Il vettore x definito sugli archi e riportato in figura è un flusso ammissibile, in quanto sono verificati sia i vincoli di conservazione di flusso che i vincoli di capacità; in particolare, nei nodi 2 e 5 entrano ed escono 7 unità di flusso, mentre nei nodi 3 e 4 ne entrano ed escono 4. Il valore del flusso è $v = 8$, pari alle unità di flusso uscenti da 1 e, equivalentemente, da quelle entranti in 6; se si aggiungesse l'arco di ritorno $(6,1)$ il suo flusso sarebbe proprio 8. Si noti che non si hanno archi *vuoti*, cioè archi con flusso nullo, mentre gli archi $(1,2)$, $(3,5)$ e $(4,6)$ sono *saturi*, cioè archi il cui flusso è uguale alla capacità.

Quindi, il problema di flusso massimo ha, rispetto al problema di flusso di costo minimo generale, due importanti caratteristiche: il vettore dei bilanci è nullo, ed il vettore dei costi è nullo tranne che in corrispondenza all' arco fittizio (t,s), in cui è negativo. Come vedremo, queste caratteristiche permettono di sviluppare algoritmi specifici molto efficienti per il problema.

2.5.1 Tagli, cammini aumentanti e condizioni di ottimo

Come nel caso del problema dei cammini minimi, consideriamo x un flusso ammissibile x di valore v, e poniamoci il problema di determinare se x sia oppure no ottimo. Se x non è ottimo, ciò significa che è possibile inviare altro flusso dall'origine alla destinazione; è intuitivo pensare che questo flusso possa essere "instradato" lungo un cammino da s a t. Sia quindi P un cammino, non necessariamente orientato, da s a t: gli archi di P possono essere partizionati nei due insiemi P^+ e P^-, detti *insieme degli archi concordi* ed *insieme degli archi discordi* di P, che contengono rispettivamente gli archi che, andando da s a t, vengono attraversati nel verso del loro orientamento e gli archi attraversati nel verso opposto a quello del loro orientamento.

Il cammino P può essere utilizzato per inviare flusso da s a t se è possibile modificare il valore del flusso su tutti i suoi archi senza perdere l'ammissibilità ed aumentando il valore v del flusso. È immediato verificare che l'unico modo in cui si può modificare il valore del flusso sugli archi del cammino senza violare i vincoli di conservazione del flusso nei nodi intermedi è quello di aumentare il flusso su tutti gli archi concordi e diminuire il flusso su tutti gli archi discordi di una stessa quantità θ. In altre parole, se x rispetta i vincoli di conservazione del flusso, allora anche $x(\theta) = x \oplus \theta P$ con

$$x_{ij}(\theta) = \begin{cases} x_{ij} + \theta, & \text{se } (i,j) \in P^+, \\ x_{ij} - \theta, & \text{se } (i,j) \in P^-, \\ x_{ij}, & \text{altrimenti.} \end{cases} \tag{2.9}$$

li rispetta per qualsiasi valore di θ, ed il valore di $x(\theta)$ è $v+\theta$; l'*operazione di composizione* $\oplus$ tra il flusso x ed il cammino P corrisponde all'invio di θ unità di flusso dall'origine alla destinazione utilizzando il cammino P. Non per tutti i valori di θ però, l'operazione di composizione produce un flusso ammissibile, in quanto possono essere violati i vincoli di capacità degli archi e di non negatività del flusso. La quantità

$$\theta(P,x) = \min\{\min\{u_{ij} - x_{ij} : (i,j) \in P^+\}, \min\{x_{ij} : (i,j) \in P^-\}\}. \quad (2.10)$$

è detta *capacità del cammino* P rispetto al flusso x, e rappresenta la massima quantità di flusso che, aggiunta agli archi concordi di P, non produce flussi maggiori delle capacità, e, sottratta agli archi discordi di P, non produce flussi negativi. Si noti che può essere $\theta(P,x) = 0$: ciò accade se almeno uno degli archi concordi è *saturo* oppure se almeno uno degli archi discordi è *vuoto*. Se invece $\theta(P,x) > 0$, P è detto un *cammino aumentante*, cioè un cammino lungo il quale può essere inviata una quantità positiva di flusso da s verso t.

Esempio 2.13:
Sia dato il grafo in figura 2.19, e si consideri il cammino (non orientato) $P = \{1,3,2,4,5,6\}$, anch'esso mostrato in figura (archi in grassetto). L'insieme degli archi concordi è $P^+ = \{(1,3),(2,4),(4,5),(5,6)\}$ mentre l'insieme degli archi discordi è $P^- = \{(2,3)\}$. La capacità di P rispetto a x è $\theta(P,x) = \min\{\min\{5-1, 8-4, 5-3, 9-7\}, \min\{3\}\} = \min\{2,3\} = 2 > 0$, e pertanto P è un cammino aumentante rispetto a x: infatti, nessun arco concorde è saturo e nessun arco discorde è vuoto.

Utilizziamo quindi P inviare altre due unità di flusso da s a t, ossia costruiamo il nuovo flusso $x' = x \oplus 2P$; applicando la definizione si ottiene $x'_{13} = 1+2 = 3$, $x'_{23} = 3-2 = 1$, $x'_{24} = 4+2 = 6$, $x'_{45} = 3+2 = 5$ e $x'_{56} = 7+2 = 9$, mentre il flusso su tutti gli altri archi è invariato. È immediato verificare che x' è un flusso ammissibile di valore $v' = 8+2 = 10$.

Si pone quindi il problema di definire un algoritmo che, dato un grafo G e un flusso ammissibile x, determini (se esiste) un cammino aumentante P rispetto ad x. A questo scopo si introduce il *grafo residuo* $G_x = (N, A_x)$ rispetto al flusso x, dove

$$A_x = \{(i,j) : (i,j) \in A, x_{ij} < u_{ij}\} \cup \{(i,j) : (j,i) \in A, x_{ji} > 0\}.$$

Il grafo residuo, cioè, contiene al più due "rappresentanti" di ciascun arco (i,j) del grafo originale: uno, orientato come (i,j), se (i,j) non è saturo e quindi può appartenere all'insieme degli archi *concordi* di un cammino aumentante, mentre l'altro, orientato in modo opposto ad (i,j), se (i,j) non è vuoto e quindi può appartenere all'insieme degli archi *discordi* di un cammino aumentante. È immediato verificare che G_x permette di ricondurre il concetto di cicli e cammini aumentanti al più usuale concetto di cicli e cammini orientati:

Lemma 2.4 *Per ogni cammino aumentante da s a t rispetto ad x in G esiste uno ed un solo cammino orientato da s a t in G_x.*

Esempio 2.14:
In figura 2.20 è mostrato il grafo residuo corrispondente all'istanza ed al flusso x di figura 2.19.

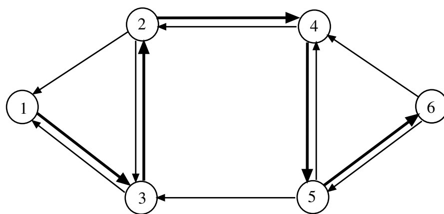

Figura 2.20: il grafo residuo per l'istanza in figura 2.19

Il cammino orientato $P = \{1, 3, 2, 4, 5, 6\}$ su G_x corrisponde al cammino aumentante su G mostrato nell'esempio precedente.

Un cammino aumentante, se esiste, può quindi essere determinato mediante una visita del grafo residuo G_x a partire da s. Se la visita raggiunge t, allora si è determinato un cammino aumentante (si noti che la visita può essere interrotta non appena questo accada), ed il flusso x non è ottimo perché il cammino aumentante permette di costruire un nuovo flusso x' di valore strettamente maggiore. Se invece al termine della visita non si è visitato t, allora x è un flusso massimo; per dimostrarlo introduciamo alcuni concetti.

Indichiamo con (N_s, N_t) un taglio, definito su G, che separa s da t, cioè un taglio per cui è $s \in N_s$ e $t \in N_t$, ed indichiamo con $A^+(N_s, N_t)$ ed $A^-(N_s, N_t)$ rispettivamente l'insieme degli archi diretti e quello degli archi inversi del taglio (si veda l'Appendice B).

Dato un flusso x, per ogni taglio (N_s, N_t) definiamo il *flusso del taglio*, $x(N_s, N_t)$, e la *capacità del taglio*, $u(N_s, N_t)$, come segue:

$$x(N_s, N_t) = \sum_{(i,j)\in A^+(N_s,N_t)} x_{ij} - \sum_{(i,j)\in A^-(N_s,N_t)} x_{ij} \ , \qquad (2.11)$$

$$u(N_s, N_t) = \sum_{(i,j)\in A^+(N_s,N_t)} u_{ij} \ . \qquad (2.12)$$

Il flusso del taglio è la quantità di flusso che attraversa il taglio (N_s, N_t) da s verso t. Il seguente teorema fornisce la relazione esistente tra il valore del flusso x, ed i flussi e le capacità dei tagli di G.

Teorema 2.6 *Per ogni flusso ammissibile x di valore v e per ogni taglio (N_s, N_t), vale:*

$$v = x(N_s, N_t) \leq u(N_s, N_t).$$

Dimostrazione La disuguaglianza deriva immediatamente da (2.11), (2.12) e dal fatto che $0 \leq x_{ij} \leq u_{ij}$ per ogni $(i, j) \in A$: infatti

$$\sum_{(i,j)\in A^+(N_s,N_t)} x_{ij} \leq \sum_{(i,j)\in A^+(N_s,N_t)} u_{ij} \qquad \text{e} \qquad - \sum_{(i,j)\in A^-(N_s,N_t)} x_{ij} \leq 0 \ .$$

L'uguaglianza si ottiene sommando tra loro, membro a membro, i vincoli di (2.8) corrispondenti ai nodi in N_t; infatti, nella sommatoria i flussi degli archi $(i,j) \in A^+(N_s,N_t)$ appaiono con coefficiente $+1$ in quanto entranti nel nodo $j \in N_t$, quelli degli archi $(i,j) \in A^-(N_s,N_t)$ appaiono con coefficiente -1 in quanto uscenti dal nodo $j \in N_t$, mentre i flussi degli archi (i,j) i cui estremi appartengono entrambi a N_t appaiono una volta con coefficiente $+1$ ed una con coefficiente -1 e quindi non contribuiscono alla somma. $\diamond$

Esempio 2.15:
Consideriamo il taglio $(N_s,N_t) = (\{1,3,5\},\{2,4,6\})$ mostrato in figura; l'insieme degli archi diretti del taglio è $A^+(N_s,N_t) = \{(1,2),(5,6)\}$, mentre quello degli archi inversi è $A^-(N_s,N_t) = \{(2,3),(4,5)\}$. Il flusso del taglio è $x(N_s,N_t) = x_{12} + x_{56} - x_{23} - x_{45} = 7+7-3-3 = 8 = v$, mentre la capacità del taglio è $u(N_s,N_t) = u_{12} + u_{56} = 7+9 = 16$.

Esercizio 2.24 *Cercare, se esiste, un taglio (N_s,N_t) nel grafo in figura 2.19 avente una capacità inferiore a 16.*

Esercizio 2.25 *Per il grafo in figura 2.19, si consideri il taglio $(N_s,N_t) = (\{1,2,5\},\{3,4,6\})$: si forniscano gli insiemi degli archi diretti e inversi del taglio si calcolino il flusso e la capacità del taglio.*

Esercizio 2.26 *Ripetere l'esercizio precedente per il taglio $(N_s,N_t) = (\{1,4,5\},\{2,3,6\})$.*

Il Teorema 2.6 mostra che, comunque si prenda un taglio che separa t da s, il valore del flusso massimo non può eccedere la capacità di tale taglio. Di conseguenza, possiamo dimostrare che x è un flusso massimo se determiniamo un taglio (N_s,N_t) la cui capacità è uguale a v. Un taglio di questo tipo è in effetti determinato dalla visita del grafo residuo G_x nel caso in cui t non viene raggiunto. Sia infatti N_s dei nodi visitati a partire da s, e $N_t = N \setminus N_s$: tutti gli archi di $A^+(N_s,N_t)$ sono saturi, altrimenti l'algoritmo avrebbe potuto visitare un ulteriore nodo, e analogamente, tutti gli archi di $A^-(N_s,N_t)$ sono vuoti. Dal Teorema 2.6 si ha allora $v = x(N_s,N_t) = u(N_s,N_t)$, ossia il flusso e la capacità del taglio coincidono: di conseguenza, x è un flusso massimo. Un'ulteriore importante conseguenza di questa relazione è che (N_s,N_t) è un *taglio di capacità minima* tra tutti i tagli del grafo che separano s da t. Abbiamo quindi dimostrato il seguente teorema:

Teorema 2.7 *(Flusso Massimo-Taglio Minimo) Il massimo valore dei flussi ammissibili su G è uguale alla minima delle capacità dei tagli di G che separano s da t.*

2.5.2 Algoritmo per cammini aumentanti

Le proprietà enunciate nel precedente paragrafo permettono di costruire un algoritmo per la determinazione di un flusso massimo. Tale algoritmo, descritto nel seguito, parte da un flusso nullo e, ad ogni passo, determina

un cammino aumentante, incrementando il valore del flusso; l'algoritmo si ferma quando non esiste più alcun cammino aumentante, restituendo quindi anche un taglio di capacità minima.

```
Procedure Cammini-Aumentanti (G, s, t, x, N_s, N_t):
  begin
    x := 0;
    while Trova-Cammino(G, s, t, x, P, θ, N_s, N_t) do
      Aumenta-Flusso(x, P, θ)
  end.
```

Procedura 2.7: Algoritmo basato su cammini aumentanti

La procedura *Trova-Cammino* cerca di determinare un cammino aumentante da s a t, dato il flusso x; se il cammino esiste *Trova-Cammino* restituisce *vero* e fornisce il cammino P e la sua capacità $\theta = \theta(P, x)$, altrimenti restituisce $falso$ e fornisce un taglio, (N_s, N_t), di capacità minima. La procedura *Aumenta-Flusso* aggiorna il flusso x inviando sul cammino P la quantità di flusso θ, ossia implementa l'operazione di composizione $x := x \oplus \theta P$.

La procedura *Trova-Cammino* è essenzialmente una visita del grafo residuo G_x a partire dall'origine s; analogamente a quanto già visto al paragrafo 2.2.2, è possibile evitare di costruire una rappresentazione di G_x modificando opportunamente la procedura *Visita* in modo che possa lavorare direttamente sulle strutture dati che descrivono il grafo originario G. Inoltre, è facile implementare *Trova-Cammino* in modo tale che contemporaneamente al cammino determini anche la sua capacità, memorizzando per ciascun nodo j raggiunto nella visita la capacità $d(j)$ dell'unico cammino da s a j sull'albero $T_s = (N_s, A_s)$ determinato fino a quel momento. Infatti, si supponga di visitare il nodo j provenendo dal nodo i: se si è usato l'arco (i, j) allora si ha che $d(j) = \min\{d(i), u_{ij} - x_{ij}\}$, mentre se si è usato l'arco (j, i) allora si ha che $d(j) = \min\{d(i), x_{ji}\}$. Per inizializzare la procedura possiamo porre $d(s) = +\infty$.

La procedura *Aumenta-Flusso* può essere implementata percorrendo il cammino in senso inverso da t a s, per mezzo della funzione predecessore, e modificando il flusso in accordo alla (2.9). Occorre però tenere conto del fatto che la funzione predecessore della procedura di visita standard non distingue l'orientamento degli archi del grafo originale: in altre parole, $p[j] = i$ non indica se si è usato l'arco (i, j) oppure l'arco (j, i) del grafo originario per raggiungere j. Per ovviare a questo inconveniente si può ad esempio porre, nella *Trova-Cammino*, $p[j] = i$ se j viene visitato a partire da i mediante l'arco (i, j), e porre invece $p[j] = -i$ se j viene visitato mediante l'arco (j, i). Alternativamente, ed in modo più generale, si può porre in $p[j]$ un puntatore all'arco (i, j) del grafo originale che viene usato per visitare j.

Esempio 2.16:
Applicando la procedura *Trova-Cammino* al grafo residuo G_x di figura 2.20 si ottiene l'albero mostrato in figura (archi in grassetto), ossia i seguenti valori dei predecessori e delle capacità:

nodi	1	2	3	4	5	6
$p[\cdot]$	*nil*	-3	1	2	4	5
$d[\cdot]$	∞	3	4	3	2	2

La procedura *Aumenta-Flusso* utilizza $p[\cdot]$ per inviare lungo il cammino $\theta = d(6) = 2$ unità di flusso: risalendo da 6 a 1 mediante la funzione predecessore si aggiunge 2 ai flussi di tutti gli archi tranne l'arco $(2, 3)$, che è un arco discorde (ciò è segnalato dal fatto che $p[2] = -3$), al cui flusso si sottrae 2; si ottiene così il nuovo flusso x' di valore $v' = 10$.

Esercizio 2.27 *Si fornisca una descrizione formale, in pseudo-codice, delle procedure* Trova-Cammino *e* Aumenta-Flusso*.*

Esercizio 2.28 *Si consideri il flusso x' determinato nell'esempio precedente e si applichi ad esso la procedura* Trova-Cammino *determinando un nuovo cammino aumentante o, se non ne esistono, un taglio di capacità (minima) pari a* 10*.*

È facile verificare che, nel caso in cui le capacità siano intere, l'algoritmo *Cammini-Aumentanti* ha complessità $O(mnU)$, con $U = \max\{u_{ij} : (i,j) \in A\}$. Infatti, se x è intero (e lo è certamente alla prima iterazione), allora θ è un valore intero, e quindi anche $x(\theta)$ sarà un flusso a valori interi: di conseguenza, ad ogni iterazione (a parte l'ultima) il valore del flusso viene aumentato di almeno un'unità. nU è maggiore della capacità del taglio $(\{s\}, N \setminus \{s\})$, e pertanto anche della capacità minima dei tagli, e di conseguenza anche del massimo valore del flusso; pertanto il numero di iterazioni sarà al più nU. Infine, ogni iterazione ha complessità $O(m)$, dovuta alla procedura di visita. Osserviamo che tale complessità è *pseudopolinomiale*, essendo U uno dei dati numerici presenti nell'input del problema.

Da questa analisi segue:

Teorema 2.8 *Se le capacità degli archi sono numeri interi, allora esiste almeno un flusso massimo intero.*

La correttezza dell'algoritmo *Cammini-Aumentanti* non dipende dal modo in cui viene determinato il cammino aumentante, ossia da come è implementato l'insieme Q nella procedura *Trova-Cammino*. La versione in cui Q è una *fila*, ossia si realizza una *visita a ventaglio* del grafo residuo, prende il nome di *algoritmo di Edmonds & Karp*. La visita a ventaglio permette di visitare ogni nodo mediante il cammino (aumentante) *più corto*, formato cioè dal minimo numero di archi; pertanto si privilegeranno inizialmente cammini aumentanti "corti", aumentando nelle iterazioni successive la loro lunghezza, ed arrivando eventualmente solo nelle ultime iterazioni ad utilizzare cammini aumentanti che passano attraverso tutti (o quasi) i nodi.

Esempio 2.17:
Un esempio di esecuzione dell'algoritmo di Edmonds & Karp è mostrato in figura 2.21. Per ogni iterazione sono mostrati lo stato iniziale del flusso ed il cammino aumentante selezionato (archi in grassetto); nell'ultima iterazione è mostrato il taglio di capacità minima determinato dall'algoritmo.

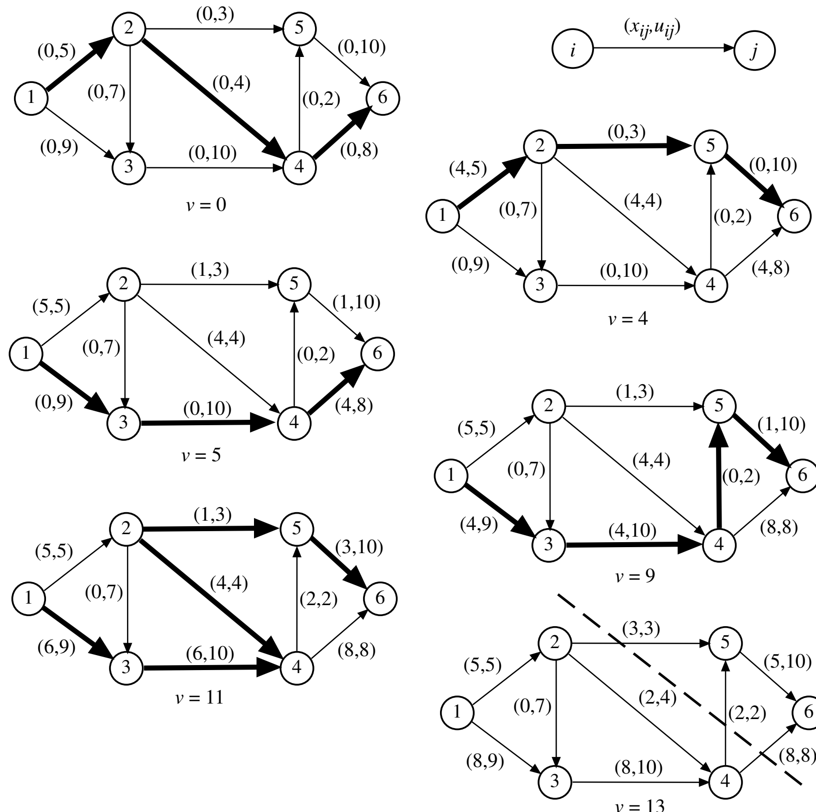

Figura 2.21: Esecuzione dell'algoritmo di Edmonds & Karp

Per l'algoritmo di Edmonds & Karp è possibile dimostrare una valutazione della complessità migliore di quella della procedura generale *Cammini-Aumentanti*; ciò è essenzialmente dovuto al seguente risultato, di cui per semplicità omettiamo la dimostrazione:

Lemma 2.5 *Nell'algoritmo di Edmonds & Karp, qualsiasi arco* (i, j) *che venga saturato o vuotato mediante un cammino di lunghezza k potrà successivamente entrare a far parte solamente di cammini aumentanti di lunghezza almeno* $k + 2$.

Questa proprietà deriva dal fatto che i cammini aumentanti "corti" sono utilizzati prima di quelli "lunghi" per via della visita a ventaglio del grafo; si veda ad esempio l'arco (2, 4) in figura 2.21. Da questo deriva il seguente:

Teorema 2.9 *L'algoritmo di Edmonds & Karp ha complessità $O(m^2 n)$.*

Dimostrazione Ad ogni iterazione dell'algoritmo viene saturato o vuotato almeno un arco, ma dal Lemma 2.5 segue immediatamente che ogni arco non può essere saturato o vuotato più di $O(n)$ volte, per cui non possono essere fatte più di $O(mn)$ iterazioni, ciascuna delle quali costa $O(m)$. ◇

Questo risultato è particolarmente interessante perché vale anche nel caso in cui le capacità non siano intere.

Esercizio 2.29 *Partendo dal flusso $x = 0$, determinare il flusso massimo da 1 a 6 sul grafo in figura 2.19, dove la capacità dell'arco $(4, 6)$ è $u_{46} = 5$, utilizzando l'algoritmo di Edmonds & Karp. Fornire per ogni iterazione l'albero della visita, il cammino aumentante, la sua capacità, il flusso e il suo valore. Fornire al termine il taglio di capacità minima.*

2.5.3 Algoritmo basato su preflussi

Un limite degli algoritmi basati su cammini aumentanti è il fatto che, ad ogni iterazione, nel caso peggiore può essere necessario esplorare tutto il grafo, senza alcuna garanzia di determinare un cammino aumentante lungo il quale sia possibile inviare una consistente quantità di flusso. Un approccio alternativo è basato sul concetto di *preflusso*.

Un preflusso è un vettore $x = [x_{ij}]$ tale che

$$\sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} \geq 0, \qquad \forall i \in N \setminus \{s, t\} ,$$

$$0 \leq x_{ij} \leq u_{ij}, \qquad \forall (i, j) \in A .$$

Quindi un preflusso soddisfa i vincoli di capacità, ma non necessariamente quelli di bilanciamento ai nodi: la quantità di flusso che arriva ad un nodo può essere maggiore di quella che esce dal nodo. Un nodo i viene detto *attivo* se il suo *eccesso*

$$e_i = \sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij}$$

è positivo; altrimenti ($e_i = 0$), i viene detto *bilanciato*.

L'idea che sta alla base dell'algoritmo basato su preflussi è di cercare di spingere flusso verso la destinazione, usando ogni volta solo informazione locale (informazione relativa al nodo in esame ed ai nodi ad esso adiacenti). A questo scopo si definisce, per ogni nodo i, una etichetta d_i con le seguenti proprietà:

$$d_t = 0 ,$$

$$d_i - d_j \leq 1, \text{ se } (i, j) \in A \text{ e } x_{ij} < u_{ij} \text{ oppure } (j, i) \in A \text{ e } x_{ji} > 0 .$$

È facile verificare che d_i è una valutazione per difetto della lunghezza (numero di archi) dei cammini aumentanti da i a t; si può cioè affermare che, se valgono tali condizioni, allora non esistono cammini aumentanti dal nodo i al pozzo t formati da meno di d_i archi. Un insieme di etichette con questa proprietà viene detto *etichettatura valida*.

Data un'etichettatura valida, un arco (i, j) è detto *ammissibile per* i se è non saturo, cioè se $x_{ij} < u_{ij}$, e se $d_i = d_j + 1$; analogamente, un arco (i, j) è detto *ammissibile per* j se è non vuoto, cioè se $x_{ij} > 0$, e se $d_j = d_i + 1$.

Sia i un nodo in attivo per x: se esiste un arco ammissibile $(i, j) \in FS(i)$, allora è possibile inviare l'eccesso, o una parte di esso, da i al nodo j che risulta, per l'etichettatura valida, "più vicino" a t, attraverso la seguente operazione di *push* lungo l'arco (i, j):

$$\theta := \min\{e_i, u_{ij} - x_{ij}\} \ , \qquad x_{ij} := x_{ij} + \theta \ .$$

Si noti che è facile aggiornare i bilanci ai nodi i e j dopo un'operazione di push: $e_i := e_i - \theta$ e $e_j := e_j + \theta$. Analogamente, se esiste un arco ammissibile $(j, i) \in BS(i)$ per un nodo attivo i, allora è possibile inviare l'eccesso, o una parte di esso, da i al nodo j che risulta, per l'etichettatura valida, più vicino a t, attraverso la seguente operazione di *push all'indietro* lungo l'arco (j, i):

$$\theta := \min\{e_i, x_{ji}\} \ , \ x_{ji} := x_{ji} - \theta \ .$$

In questo caso, l'aggiornamento dei bilanci ai nodi i e j è: $e_i := e_i + \theta$ e $e_j := e_j - \theta$. In entrambi i casi, essendo gli archi ammissibili, si ha $\theta > 0$.

Supponiamo ora che il nodo attivo i non abbia archi incidenti che siano ammissibili. Tale situazione avviene quando:

- per ogni arco $(i, j) \in FS(i)$ si ha $x_{ij} = u_{ij}$ (arco saturo), oppure $d_i < d_j + 1$;
- per ogni arco $(j, i) \in BS(i)$ si ha $x_{ji} = 0$ (arco vuoto), oppure $d_i < d_j + 1$.

In altri termini, possiamo affermare che non esistono cammini aumentanti da i a t formati da d_i archi; quindi l'etichetta di i può essere incrementata attraverso la seguente operazione di *relabel*:

$$d_i := 1 + \min\{d_j : (i, j) \in FS(i) \text{ con } x_{ij} < u_{ij} \ , \ (j, i) \in BS(i) \text{ con } x_{ji} > 0\}.$$

L'operazione di relabel rende ammissibile almeno un arco incidente in i (quello per cui si è ottenuto il valore minimo). Si noti che la massima lun-ghezza di un cammino aumentante è $n - 1$; pertanto, per ogni nodo i con $d_i \geq n$, si può affermare che non esistono cammini aumentanti da esso a t.

Si può dimostrare che, scegliendo opportunamente il preflusso iniziale, è sempre possibile effettuare o un'operazione di push oppure un'operazione

di relabel; ossia, non può mai darsi il caso che un nodo i sia attivo, tutti gli archi della $FS(i)$ siano saturi e tutti gli archi della $BS(i)$ siano vuoti. Inoltre, si può facilmente verificare che, se da un certo nodo i non si possono effettuare operazioni di push, allora l'applicazione di un'operazione di relabel del nodo i a partire da un'etichettatura valida produce una nuova etichettatura anch'essa valida.

Presentiamo adesso l'algoritmo basato su preflussi, che utilizza le operazioni di push e relabel per risolvere il problema del flusso massimo.

La procedura *Etichettatura-Valida* costruisce un'etichettatura valida d, ad esempio ponendo d_i uguale alla lunghezza del cammino minimo, in termini di numero di archi, da i a t. Ciò può essere facilmente ottenuto mediante una visita a ventaglio "all'indietro", ossia una visita a partire dal nodo t su $G' = (N, A')$ con $A' = \{(j,i) : (i,j) \in A\}$, in cui Q è implementato come una fila.

Procedure *Preflow-Push* (G, s, t, x):
  begin
    $x := 0$; $x_{sj} := u_{sj}$, $\forall (s,j) \in FS(s)$; *Etichettatura-Valida*(G, d); $d_s := n$;
    repeat
      Seleziona un nodo v con $e_v > 0$;
      if $\exists (v,j)$ con $x_{vj} < u_{vj}$ e $d_v = d_j + 1$ **then**
        begin
          $\theta := \min\{e_v, u_{vj} - x_{vj}\}$;
          $x_{vj} := x_{vj} + \theta$; $e_j := e_j + \theta$; $e_v := e_v - \theta$
        end
        else if $\exists (i,v)$ con $x_{iv} > 0$ e $d_v = d_i + 1$ **then**
            begin
              $\theta := \min\{e_v, x_{iv}\}$;
              $x_{iv} := x_{iv} - \theta$; $e_i := e_i + \theta$; $e_v := e_v - \theta$
            end
          else
            begin
              $d' := \min\{d_j : (v,j) \in FS(v), x_{vj} < u_{vj}\}$;
              $d'' := \min\{d_i : (i,v) \in BS(v), x_{iv} > 0\}$;
              $d_v := 1 + \min\{d', d''\}$
            end
    until $e_i = 0$, $\forall i$
  end.

Procedura 2.8: Algoritmo basato su preflussi

Se è possibile effettuare un'operazione di push, l'algoritmo sposta flusso da un nodo attivo v ad un nodo j se j è "più vicino" a t rispetto a v, secondo l'etichettatura valida d. Se non sono possibili operazioni di push, allora d_v è una sottostima della reale distanza di t da v lungo cammini aumentanti, e l'etichetta viene incrementata mediante un'operazione di relabel.

Un modo figurato per visualizzare il significato delle etichette è quello di considerarle come l'altezza dei nodi rispetto ad un piano comune: il flusso va dai nodi "più in alto" a quelli "più in basso", ma scendendo di un solo livello. Se un nodo ha un eccesso di flusso che non riesce a smaltire, viene portato al livello superiore del più in basso dei suoi vicini (raggiungibili attraverso archi non saturi o non vuoti) in modo da consentirgli di diminuire il suo sbilanciamento.

Quando l'algoritmo termina, oltre ad un flusso massimo riporta anche un taglio (N_s, N_t) di capacità minima; in particolare, fanno parte di N_s tutti quei nodi che, alla terminazione dell'algoritmo, hanno un'etichetta con valore maggiore od uguale a n.

Esempio 2.18:
Nella tabella 2.1 è riportata una descrizione di tutte le operazione compiute dall'algoritmo *Preflow-Push* per risolvere l'istanza di figura 2.21, a partire dall'etichettatura valida $d = [6, 2, 2, 1, 1, 0]$. Una descrizione grafica delle operazioni è anche mostrata in figura 2.22; per brevità, nella figura non sono mostrate tutte le iterazioni, ma sono evidenziati gli archi convolti in operazioni di push.

nodo	oper.	arco	θ	correzioni			
2	push	(2,4)	3	$e_x(2) = 2$	$x_{24} = 3$	$e_x(4) = 3$	
2	push	(2,5)	2	$e_x(2) = 0$	$x_{25} = 2$	$e_x(5) = 2$	
3	push	(3,5)	9	$e_x(3) = 0$	$x_{35} = 9$	$e_x(5) = 11$	
4	push	(4,6)	3	$e_x(4) = 0$	$x_{46} = 3$	$v_t = 3$	
5	push	(5,6)	8	$e_x(5) = 3$	$x_{56} = 8$	$v_t = 11$	
5	relabel			$d_5 =$	$d_4 + 1$	$= 2$	
5	push	(5,4)	2	$e_x(5) = 1$	$x_{54} = 2$	$e_x(4) = 2$	
4	push	(4,6)	2	$e_x(4) = 0$	$x_{46} = 5$	$v_t = 13$	
5	relabel			$d_5 =$	$d_2 + 1$	$= 3$	
5	push	(2,5)	1	$e_x(5) = 0$	$x_{25} = 1$	$e_x(2) = 1$	
2	relabel			$d_2 =$	$d_3 + 1$	$= 3$	
2	push	(2,3)	1	$e_x(2) = 0$	$x_{23} = 1$	$e_x(3) = 1$	
3	relabel			$d_3 =$	$d_2 + 1$	$= 4$	
3	push	(2,3)	1	$e_x(3) = 0$	$x_{23} = 0$	$e_x(2) = 1$	
2	relabel			$d_2 =$	$d_5 + 1$	$= 4$	
2	push	(2,5)	1	$e_x(2) = 0$	$x_{25} = 2$	$e_x(5) = 1$	
5	relabel			$d_5 =$	$d_2 + 1$	$= 5$	
5	push	(2,5)	1	$e_x(5) = 0$	$x_{25} = 1$	$e_x(2) = 1$	
2	relabel			$d_2 =$	$d_3 + 1$	$= 5$	
2	push	(2,3)	1	$e_x(2) = 0$	$x_{23} = 1$	$e_x(3) = 1$	
3	relabel			$d_3 =$	$d_2 + 1$	$= 6$	
3	push	(2,3)	1	$e_x(3) = 0$	$x_{23} = 0$	$e_x(2) = 1$	
2	relabel			$d_2 =$	$d_5 + 1$	$= 6$	
2	push	(2,5)	1	$e_x(2) = 0$	$x_{25} = 2$	$e_x(5) = 1$	
5	relabel			$d_5 =$	$d_2 + 1$	$= 7$	
5	push	(3,5)	1	$e_x(5) = 0$	$x_{35} = 8$	$e_x(3) = 1$	
3	relabel			$d_3 =$	$d_2 + 1$	$= 7$	
3	push	(3,1)	1	$e_x(3) = 0$	$x_{13} = 8$	$v_s = 13$	STOP

Tabella 2.1: Esecuzione dell'algoritmo *Preflow-Push*

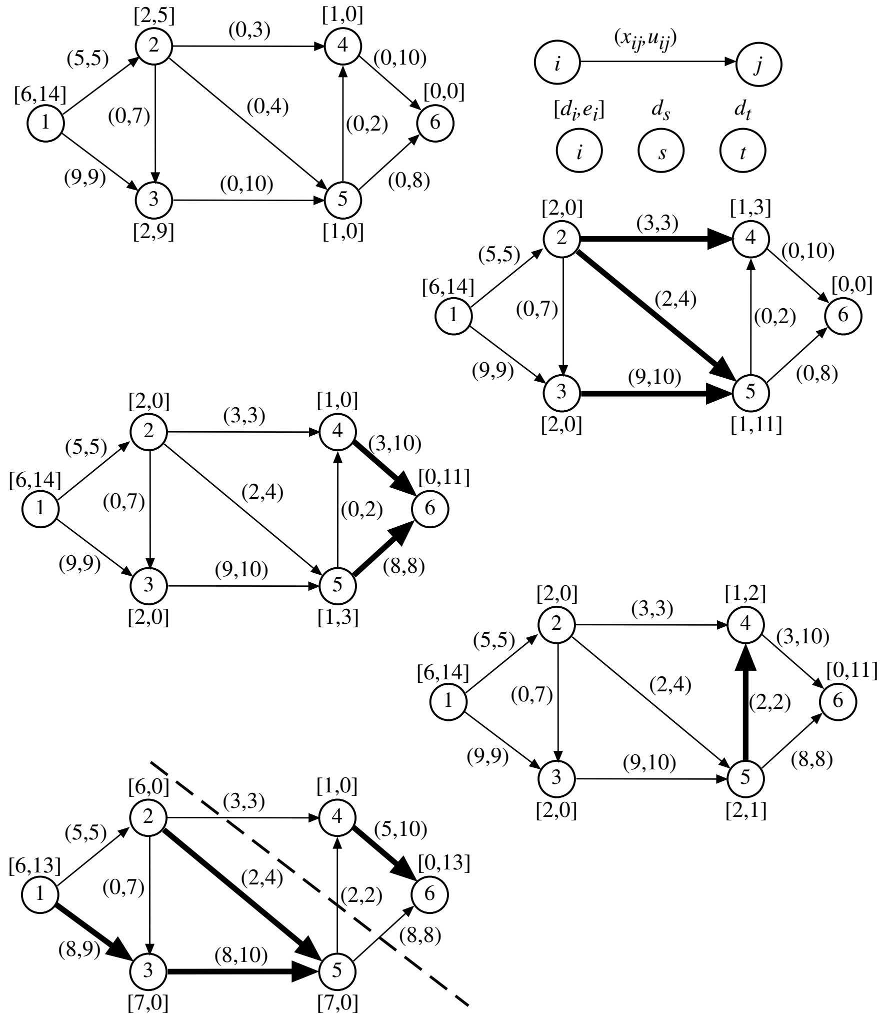

Figura 2.22: Esecuzione dell'algoritmo *Preflow-Push*

L'esempio precedente illustra alcune peculiarità dell'algoritmo: inizialmente viene introdotta nel grafo una quantità di flusso maggiore o uguale del flusso massimo, corrispondente alla capacità del taglio $(\{s\},\ N \setminus \{s\})$. Una volta che tutto il flusso possibile è giunto a t, quello eventualmente in eccesso viene riportato a s: per questo, le etichette dei nodi appartenenti a N_s nel taglio ottimo devono crescere ad un valore maggiore od uguale a n. L'algoritmo esegue allora una sequenza di operazioni di push e relabel mediante le quali i nodi in N_s si inviano ripetutamente le unità di flusso in eccesso al solo scopo di far aumentare le loro etichette fino ad un valore che consenta di reinstradare il flusso fino ad s. Nell'esempio questa fase dell'algoritmo inizia alla nona iterazione e coinvolge i nodi 2, 3 e 5 e gli archi $(2,3)$, $(2,5)$ e $(3,5)$.

Esistono comunque modi per sveltire l'algoritmo, evitando almeno par-

zialmente questa fase di reinstradamento del flusso in eccesso.

Essi si basano sull'evento che vi sia un *salto di etichettatura*, cioè che esista un valore intero $k < n$ per cui nessun nodo ha etichetta uguale a k. In tal caso, a tutti i nodi i con etichetta $k < d_i < n$ può essere assegnata una etichetta $d_i = n + 1$.

Questo capita nell'esempio precedente, subito dopo l'operazione di relabel del nodo 3 in cui $d_3 = 4$. Infatti, in quel momento, il vettore delle etichette è $d = [6, 3, 4, 1, 3, 0]$; si ha un salto di etichettatura per $k = 2$ e si può porre $d_2 = d_3 = d_5 = 7$, risparmiando 14 operazioni e potendo reinstradare immediatamente l'unità di flusso in eccesso al nodo sorgente.

È possibile dimostrare che l'algoritmo basato su preflussi fornisce la soluzione ottima con complessità in tempo pari a $O(n^2m)$; se il nodo attivo v viene selezionato secondo criteri specifici, la complessità può essere ridotta a $O(n^3)$. L'algoritmo risulta anche essere molto efficiente in pratica, e se ne possono dare implementazioni parallele in ambiente asincrono; per maggiori approfondimenti su questi temi si rinvia alla letteratura indicata ed a corsi successivi dell'area di Ricerca Operativa.

Esercizio 2.30 *Si discuta come modificare la procedura* Preflow-Push *per risolvere le due varianti del problema di flusso massimo con più sorgenti/pozzi presentate al paragrafo 2.5.4 senza costruire esplicitamente il grafo aumentato G' (si noti che nella prima variante gli archi uscenti dalla "super-radice" s hanno capacità infinita).*

2.5.4 Flusso massimo con più sorgenti/pozzi

Una generalizzazione del problema del flusso massimo è quella in cui si ha un insieme S di nodi sorgente ed un insieme T di nodi pozzo (entrambe non vuoti) e si vuole individuare il massimo flusso che può essere spedito dai nodi sorgente ai nodi pozzo. Questo problema può essere facilmente risolto applicando un qualunque algoritmo per il flusso massimo ad un grafo ampliato $G' = (N', A')$ con $N' = N \cup \{s, t\}$ e $A' = A \cup \{(s, j) : j \in S\} \cup \{(i, t) : i \in T\}$; s e t sono la "super-sorgente" ed il "super-pozzo", collegati rispettivamente a tutti i nodi in S e T con archi a capacità infinita. Questa trasformazione è analoga a quella illustrata in figura 2.4.

È possibile modificare l'algoritmo *Cammini-Aumentanti* in modo tale che risolva direttamente il problema più generale senza la necessità di costruire esplicitamente il grafo ampliato G'. Per questo è sufficiante che la procedura *Trova-Cammino* implementi una visita a partire dall'insieme di nodi S invece che dal singolo nodo s, ossia inizializzando $Q = S$, come visto nel paragrafo 2.2.2; la visita è interrotta non appena si raggiunga un qualsiasi nodo in T, nel qual caso si è determinato un cammino aumentante da una delle sorgenti ad uno dei pozzi, oppure quando Q è vuoto, e quindi si è determinato un taglio saturo che separa *tutte* le sorgenti da *tutti* i pozzi.

Esercizio 2.31 *Si dia una descrizione formale, in pseudo-codice, dell'algoritmo per risolvere il problema del flusso massimo tra un insieme di sorgenti S ed un insieme di destinazioni T su una rete G con capacità u sugli archi.*

Un'ulteriore generalizzazione del problema è quella in cui ciascuna sorgente $i \in S$ e ciascun pozzo $i \in T$ ha una *capacità finita* u_i, ossia una massima quantità di flusso che può immettere nella/prelevare dalla rete. Anche questo problema può essere risolto applicando un algoritmo per il flusso massimo alla rete ampliata G', con l'unica differenza che la capacità degli archi (s, i) e (i, t) viene posta a u_i e non a $+\infty$. Questa versione del problema permette di *determinare se un problema di (MCF) ammette soluzione ammissibile*, ossia se esiste oppure no un flusso che rispetta tutti i vincoli di (2.1). Basta infatti definire $S = \{i : b_i < 0\}$ e $T = \{i : b_i > 0\}$, ponendo $u_i = -b_i$ per $i \in S$ e $u_i = b_i$ per $i \in T$; se il flusso massimo "satura" tutte le sorgenti (tutti gli archi (s, i)), e, di conseguenza, tutti i pozzi (tutti gli archi (i, t)), allora tale flusso è ammissibile per il problema del flusso di costo minimo; altrimenti non esistono flussi ammissibili.

Anche in questo caso è possibile modificare l'algoritmo per cammini aumentanti in modo tale che risolva la versione più generale del problema senza costruire esplicitamente la rete G'. Per questo occorre però mantenere traccia del valore x_i (il flusso sugli archi (s, i) e (i, t)) del flusso già immesso dalla sorgente/prelevato dal pozzo i nel flusso corrente x, e mantenere gli insiemi S_x e T_x delle sorgenti/pozzi che possono ancora immettere/prelevare flusso (tali che gli archi $(s, i)/(i, t)$ non sono saturi). La procedura *Trova-Cammino* implementa una visita a partire da S_x (ponendo $Q = S_x$), che viene interrotta non appena si raggiunga un qualsiasi nodo in T_x, nel qual caso si è determinato un cammino aumentante da una delle sorgenti $i \in S_x$ ad uno dei pozzi $j \in T_x$, oppure quando Q è vuoto. La procedura *Aumenta-Flusso* deve tener conto della massima quantità di flusso che i può ancora immettere nella rete e che j può ancora prelevare dalla rete (le capacità residue di (s, i) e (j, t)), ed aggiornare gli insiemi S_x e T_x qualora i e/o j si "saturino".

Esercizio 2.32 *Si dia una descrizione formale, in pseudo-codice, dell'algoritmo per determinare se una rete G con vettore di capacità u sugli archi e con vettore di bilanci b sui nodi ammette un flusso ammissibile.*

Esercizio 2.33 *Si determini un flusso ammissibile per la rete in figura 2.1 utilizzando l'algoritmo dell'esercizio precedente.*

2.6 Il problema di flusso di costo minimo

Nel paragrafo 2.1 abbiamo introdotto il problema di flusso di costo minimo; ricordiamo che i dati del problema consistono in una rete $G = (N, A)$ in cui ad ogni arco $(i, j) \in A$ sono associati un costo c_{ij} ed una capacità superiore

u_{ij} (la capacità inferiore si assume pari a 0), e ad ogni nodo $i \in N$ è associato il valore b_i.

Il problema del flusso di costo minimo è più generale del problema dei cammini minimi e del problema di flusso massimo: infatti, entrambe questi problemi possono essere formulati come particolari problemi di flusso di costo minimo. Si noti che il problema dei cammini minimi e quello di flusso massimo presentano caratteristiche distinte, che sono contemporaneamente presenti nel problema di flusso di costo minimo: nel problema dei cammini minimi gli archi hanno associati costi ma non capacità, mentre nel problema di flusso massimo gli archi hanno associate capacità ma non costi. In entrambi i casi, inoltre, la struttura delle domande/offerte dei nodi è molto particolare. In effetti, gli algoritmi per il problema di flusso di costo minimo spesso fanno uso, al loro interno, di algoritmi per i cammini minimi o per il flusso massimo.

2.6.1 Cammini, cicli aumentanti e condizioni di ottimo

Il primo passo per lo sviluppo degli algoritmi consiste nello studio delle condizioni di ottimalità per il problema, ossia delle proprietà che consentono di verificare se una data soluzione è ottima. Ciò richiede l'introduzione di alcuni importanti concetti.

Uno *pseudoflusso* è un vettore $x \in \mathbb{R}^m$ tale che $0 \leq x \leq u$, ossia che rispetta tutti i vincoli di capacità sugli archi. Definiamo *sbilanciamento* di un nodo i rispetto ad x la quantità

$$e_x(i) = \left(\sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} \right) - b_i .$$

Chiameremo

$$O_x = \{ i \in N : e_x(i) > 0 \}, \quad D_x = \{ i \in N : e_x(i) < 0 \},$$

rispettivamente l'insieme dei nodi con *eccedenza di flusso*, e quello con *difetto di flusso*: se $D_x = O_x = \emptyset$, ossia tutti i nodi sono *bilanciati*, il vettore x rispetta tutti i vincoli di conservazione di flusso ed è pertanto un flusso ammissibile. Nel seguito indicheremo con e_x il *vettore degli sbilanciamenti* del vettore x, e con

$$g(x) = \sum_{i \in O_x} e_x(i) \quad \left(= - \sum_{j \in D_x} e_x(j) \right) ,$$

il suo *sbilanciamento complessivo*; x è un flusso ammissibile se e solo se $g(x) = 0$.

Dato un cammino P, non necessariamente orientato, tra una qualunque coppia di nodi s e t del grafo, consideriamo il verso di P come quello che va da s a t; gli archi del cammino sono partizionati nei due insiemi P^+ e P^-, rispettivamente degli archi concordi e discordi col verso del cammino. Un cammino si dirà *aumentante* se la sua capacità $\theta(P, x)$, definita in (2.10), è positiva. Dato uno pseudoflusso x, è possibile inviare una quantità di flusso $0 < \theta \leq \theta(P, x)$ lungo P mediante l'*operazione di composizione* definita in (2.9), ottenendo un nuovo pseudoflusso $x(\theta) = x \oplus \theta P$ tale che

$$e_{x(\theta)}(i) = \begin{cases} e_x(s) - \theta, & \text{se } i = s, \\ e_x(t) + \theta, & \text{se } i = t, \\ e_x(i), & \text{altrimenti.} \end{cases}$$

In altre parole, inviare flusso lungo un cammino aumentante modifica solamente lo degli estremi del cammino, mentre lo sbilanciamento di tutti gli altri nodi rimane invariato.

Un caso particolare di cammino aumentante è quello in cui $s = t$, ossia il cammino è un *ciclo* su cui è arbitrariamente fissato un verso di percorrenza. Chiaramente, inviare flusso lungo un ciclo aumentante non modifica lo sbilanciamento di alcun nodo. Di conseguenza, se x in particolare è un flusso *ammissibile*, allora ogni flusso $x(\theta) = x \oplus \theta C$ per $0 \leq \theta \leq \theta(C, x)$ è ancora ammissibile: l'operazione di composizione con un *ciclo aumentante* C permette, a partire da un flusso ammissibile x, di costruire un diverso flusso ammissibile.

Il *costo* di un cammino (o ciclo) P, che indicheremo con $c(P)$, è il costo di un'unità di flusso inviata lungo P secondo il verso fissato, ossia

$$c(P) = \sum_{(i,j)\in P^+} c_{ij} - \sum_{(i,j)\in P^-} c_{ij} \ ;$$

è immediato verificare che

$$cx(\theta) = c(x \oplus \theta P) = cx + \theta c(P) \ . \tag{2.13}$$

Per determinare cicli e/o cammini aumentanti si può usare il *grafo residuo* $G_x = (N, A_x)$ rispetto allo pseudoflusso x, come è stato fatto per il flusso massimo: per ogni arco $(i, j) \in A$ si pone (i, j) in A_x, con costo $c'_{ij} = c_{ij}$, se e solo se $x_{ij} < u_{ij}$, e si pone (j, i) in A_x, con costo $c'_{ji} = -c_{ij}$, se e solo se $x_{ij} > 0$. È immediato verificare che vale la seguente generalizzazione del Lemma 2.4:

Lemma 2.6 *Comunque si fissino s e t, per ogni cammino aumentante da s a t rispetto ad x in G esiste uno ed un solo cammino orientato da s a t in G_x, ed i due cammini hanno lo stesso costo.*

Cammini e cicli aumentanti sono gli "strumenti base per costruire i flussi", come dimostra il seguente teorema:

Teorema 2.10 *Siano dati due qualunque pseudoflussi x' ed x'': allora esistono $k \leq n+m$ cammini o cicli aumentanti (semplici) per x', $P_1, \ldots, P_k$, di cui al più m sono cicli, tali che $x^1 = x'$, $x^{i+1} = x^i \oplus \theta_i P_i$, per $i = 1, \ldots, k$, $x^{k+1} = x''$, dove $0 < \theta_i = \theta(P_i, x^i)$, $i = 1, \ldots, k$. In particolare, tutti i cammini aumentanti hanno come estremi nodi in cui lo sbilanciamento di x' è diverso dallo sbilanciamento di x'', per cui se x' ed x'' hanno lo stesso vettore di sbilanciamenti, allora tutti i P_i sono cicli.*

Dimostrazione La dimostrazione è costruttiva: manteniamo uno pseudoflusso x, inizialmente pari ad x', ed in un numero finito di passi lo rendiamo uguale ad x'' utilizzando cammini e cicli aumentanti per x'. Per questo definiamo il grafo $\bar{G}_x = (N, \bar{A}_x^+ \cup \bar{A}_x^-)$, dove $\bar{A}_x^+ = \{(i,j) : x''_{ij} > x_{ij}\}$ e $\bar{A}_x^- = \{(j,i) : x''_{ij} < x_{ij}\}$. $\bar{G}_x$ "descrive" la differenza tra x ed x''; infatti, è immediato verificare che $\bar{A}_x^+ = \bar{A}_x^- = \emptyset$ se e solo se $x'' = x$. Ad ogni arco $(i,j) \in \bar{A}_x^+$ associamo la quantità $d_x(i,j) = x''_{ij} - x_{ij} > 0$, e, analogamente, ad ogni arco $(j,i) \in \bar{A}_x^-$ associamo la quantità $d_x(j,i) = x_{ij} - x''_{ij} > 0$. Definiamo poi gli insiemi

$$\bar{O}_x = \{i \in N : e_x(i) > e_{x''}(i)\} \qquad \bar{D}_x = \{i \in N : e_x(i) < e_{x''}(i)\} .$$

$\bar{O}_x$ è l'insieme dei nodi che hanno sbilanciamento rispetto a x maggiore dello sbilanciamento rispetto a x'', mentre i nodi in $\bar{D}_x$ sono quelli in cui avviene l'opposto. È facile verificare che

- $\bar{O}_x = \bar{D}_x = \emptyset$ se e solo se x ed x'' hanno lo stesso vettore di sbilanciamento;
- tutti i nodi in $\bar{O}_x$ hanno almeno un arco uscente in $\bar{G}_x$, tutti i nodi in $\bar{D}_x$ hanno almeno un arco entrante in $\bar{G}_x$, mentre tutti i nodi in $N \setminus (\bar{O}_x \cup \bar{D}_x)$ o non hanno né archi entranti né archi uscenti oppure hanno sia almeno un arco entrante che almeno un arco uscente.

Utilizzando $\bar{G}_x$ è possibile costruire iterativamente i cicli e cammini richiesti. Se $\bar{A}_x^+ = \bar{A}_x^- = \emptyset$, ossia $x = x''$, il procedimento termina, altrimenti consideriamo gli insiemi $\bar{O}_x$ e $\bar{D}_x$: se $\bar{O}_x \neq \emptyset$ si seleziona un nodo $s \in \bar{O}_x$, altrimenti, per costruire un ciclo, si seleziona un qualsiasi nodo s che abbia almeno un arco uscente (e quindi almeno uno entrante). Si visita quindi $\bar{G}_x$ a partire da s, che ha sicuramente almeno un arco uscente; siccome ogni nodo tranne al più quelli in $\bar{D}_x$ ha almeno un arco uscente, in un numero finito di passi la visita:

- o raggiunge un nodo $t \in \bar{D}_x$;
- oppure torna su un nodo già precedentemente visitato.

Nel primo caso si determina un cammino (semplice) P in $\bar{G}_x$ tra un nodo $s \in \bar{O}_x$ ed un nodo $t \in \bar{D}_x$; su questo cammino viene inviata una quantità di flusso pari a

$$\theta = \min\{\min\{d_x(i,j) : (i,j) \in P\}, e_x(s) - e_{x''}(s), e_{x''}(t) - e_x(t)\} .$$

Altrimenti si determina un ciclo (semplice) C in $\bar{G}_x$; su questo ciclo viene inviata una quantità di flusso pari a $\theta = \min\{d_x(i,j) : (i,j) \in C\}$.

In entrambi i casi si ottiene un nuovo pseudoflusso x "più simile" ad x'' del precedente, in quanto $d_x(i,j)$ per ogni (i,j) di P (o C) diminuisce della quantità $\theta > 0$. In particolare, se si determina un ciclo, si avrà $d_x(i,j) = 0$ per almeno un $(i,j) \in C$, e quindi tale arco non comparirà più in $\bar{G}_x$; se invece si determina un cammino allora si avrà che o $d_x(i,j) = 0$ per almeno un $(i,j) \in P$, oppure lo sbilanciamento rispetto ad x di almeno uno tra s e t diventa pari allo sbilanciamento rispetto ad x'', e quindi almeno uno tra s e t non comparirà più in $\bar{O}_x$ o in $\bar{D}_x$.

Si noti che sugli archi di $\bar{A}_{x'}^+$ il flusso può solo aumentare, mentre sugli archi di $\bar{A}_{x'}^-$ il flusso può solo diminuire; siccome il flusso su (i,j) non viene più modificato non appena $x_{ij} = x''_{ij}$, nessun "nuovo" arco può essere creato in $\bar{G}_x$. Pertanto, ogni grafo $\bar{G}_x$ è un

sottografo del grafo residuo iniziale $G_{x'}$, e quindi un qualunque cammino o ciclo che viene utilizzato è aumentante rispetto allo pseudoflusso iniziale x'.

Siccome ad ogni passo o si cancella almeno un arco da $\bar{G}_x$ o si cancella almeno un nodo da $\bar{O}_x \cup \bar{D}_x$, in al più $n + m$ passi tutti gli archi di $\bar{G}_x$ vengono cancellati. $\diamond$

Il Teorema 2.10 ci consente di caratterizzare gli pseudoflussi (e quindi i flussi) "ottimi". Definiamo *minimale* uno pseudoflusso x che abbia costo minimo tra tutti gli pseudoflussi aventi lo stesso vettore di sbilanciamento e_x; si noti che la soluzione ottima del problema di flusso di costo minimo è un flusso ammissibile minimale, avendo costo minimo tra tutti gli (pseudo)flussi con $e_x = 0$. Vale il seguente lemma:

Lemma 2.7 *Uno pseudoflusso (flusso ammissibile) x è minimale (ottimo) se e solo se non esistono cicli aumentanti rispetto ad x il cui costo sia negativo.*

Dimostrazione Chiaramente, se esiste un ciclo aumentante C rispetto ad x il cui costo $c(C)$ è negativo, allora x non è minimale: per ogni $0 < \theta \leq \theta(C, x)$, lo pseudoflusso $x(\theta) = x \oplus \theta C$ ha lo stesso vettore di sbilanciamento di x, ma $cx(\theta) < cx$ (si veda (2.13)).

Viceversa, sia x uno pseudoflusso tale che non esistono cicli aumentanti di costo negativo rispetto ad x, e supponiamo che x non sia minimale, ossia che esista uno pseudoflusso x' con lo stesso vettore di sbilanciamento tale che $cx' < cx$. Per il Teorema 2.10 ha

$$x' = x \oplus \theta_1 C_1 \oplus \ldots \oplus \theta_k C_k \ ,$$

dove C_i sono cicli aumentanti per x, ma siccome tutti i θ_i sono numeri positivi, $cx' < cx$ e (2.13) implicano che $c(C_i) < 0$ per un qualche i, il che contraddice l'ipotesi. $\diamond$

Nei prossimi paragrafi vedremo come la teoria appena sviluppata può essere utilizzata per costruire algoritmi per la soluzione del problema del flusso di costo minimo.

2.6.2 Algoritmo basato su cammini minimi successivi

L'algoritmo dei *cammini minimi successivi* mantiene ad ogni passo uno pseudoflusso *minimale* x, e determina un cammino aumentante *di costo minimo* tra un nodo $s \in O_x$ ed un nodo $t \in D_x$ per diminuire, al minor costo possibile, lo sbilanciamento di x. L'uso di cammini aumentanti di costo minimo permette di conservare la minimalità degli pseudoflussi:

Teorema 2.11 *Sia x uno pseudoflusso minimale, e sia P un cammino aumentante rispetto a x avente costo minimo tra tutti i cammini che uniscono un dato nodo $s \in O_x$ ad un dato nodo $t \in D_x$: allora, comunque si scelga $\theta \leq \theta(P, x)$, $x(\theta) = x \oplus \theta P$ è uno pseudoflusso minimale.*

Dimostrazione Fissato $\theta \leq \theta(P, x)$, sia x' un qualsiasi pseudoflusso con vettore di sbilanciamento $e_{x(\theta)}$. Il Teorema 2.10 mostra che esistono k cammini aumentanti $P_1, \ldots, P_k$ da s a t (in quanto s e t sono gli unici nodi in cui e_x ed $e_{x(\theta)}$ differiscono) e h cicli aumentanti $C_1, \ldots, C_h$ rispetto ad x tali che

$$x' = x \oplus \theta_1 P_1 \oplus \ldots \oplus \theta_k P_k \oplus \theta_{k+1} C_1 \oplus \ldots \oplus \theta_{k+h} C_h \; .$$

In più, deve sicuramente essere $\theta_1 + \ldots + \theta_k = \theta$. Siccome x è minimale, ciascuno degli h cicli aumentanti deve avere costo non negativo; inoltre, siccome P ha costo minimo tra tutti i cammini aumentanti tra s e t, si ha $c(P) \leq c(P_i)$, $i = 1, \ldots, k$. Di conseguenza

$$cx' = cx + \theta_1 c(P_1) + \ldots + \theta_k c(P_k) + \theta_{k+1} c(C_1) + \ldots + \theta_{k+h} c(C_h) \geq cx + \theta c(P) = cx(\theta) \; ,$$

e quindi $x(\theta)$ è minimale. ◇

Con un opportuna scelta di θ, l'operazione di composizione tra lo pseudoflusso x ed il cammino P permette di diminuire lo sbilanciamento complessivo: infatti, è immediato verificare che per

$$\theta = \min\{\theta(P, x), e_x(s), -e_x(t)\} > 0 \tag{2.14}$$

(si ricordi che $e_x(s) > 0$ e $e_x(t) < 0$), $x(\theta)$ è uno pseudoflusso (minimale) con sbilanciamento complessivo $g(x(\theta)) = g(x) - \theta < g(x)$. Questa scelta di θ corrisponde alla maggior diminuzione possibile dello sbilanciamento complessivo corrispondente al cammino P ed allo pseudoflusso x.

L'algoritmo termina se lo pseudoflusso (minimale) corrente ha sbilanciamento nullo, ossia è un flusso ottimo, oppure se non esistono più cammini aumentanti tra nodi in O_x e nodi in D_x, nel qual caso il problema non ha soluzioni ammissibili.

```
Procedure Cammini-Minimi-Successivi (G, c, b, u, x, caso):
  begin
    Inizializza(x); caso := "ottimo";
    while g(x) ≠ 0 and caso ≠ "vuoto" do
      if Trova-Cammino-Minimo(G_x, O_x, D_x, P, θ)
        then Aumenta-Flusso(x, P, θ)
        else caso := "vuoto"
  end.
```

Procedura 2.9: *Algoritmo basato su cammini minimi successivi*

La procedura *Inizializza* costruisce uno pseudoflusso x minimale: un semplice modo per implementare tale procedura è quello di porre

$$x_{ij} = \begin{cases} 0, & \text{se } c_{ij} \geq 0, \\ u_{ij}, & \text{altrimenti.} \end{cases}$$

In tal modo i costi degli archi in G_x sono tutti non negativi, non esistono quindi cicli orientati in G_x (cicli aumentanti rispetto ad x in G) di costo negativo, per cui x è minimale. Si noti che questa fase di inizializzazione richiede che non esistano archi con costo negativo e capacità infinita.

La procedura *Trova-Cammino-Minimo* determina un cammino aumentante di costo minimo P da un qualsiasi nodo $s \in O_x$ a un qualsiasi nodo $t \in D_x$. Un possibile modo per implementare questa procedura è di risolvere un problema di albero dei cammini minimi con insieme di nodi radice O_x (si veda il paragrafo 2.3.7) su G_x; in altri termini, se $|O_x| > 1$ si aggiunge a G_x un nodo "radice" r collegato a tutti i nodi in O_x con archi a costo nullo, e poi si risolve un problema di albero dei cammini minimi di radice r sul grafo così ottenuto (altrimenti basta usare come radice l'unico nodo in O_x). La procedura determina sicuramente un albero dei cammini minimi: infatti x è minimale, e quindi non esistono cicli negativi in G_x. Una volta calcolato l'albero dei cammini minimi, si seleziona un qualsiasi nodo $t \in D_x$ (ad esempio, quello con l'etichetta d_t minima, corrispondente al più corto tra tutti i cammini minimi) e si esamina il valore della sua etichetta. Se $d_t = \infty$, allora non esiste alcun cammino aumentante tra qualsiasi nodo $s \in O_x$ e t, *Trova-Cammino-Minimo* restituisce $falso$ e di conseguenza *Cammini-Minimi-Successivi* restituisce $caso$ = "vuoto"; infatti, in questo caso non esiste nessuna soluzione ammissibile per il problema di flusso di costo minimo (si veda il paragrafo 2.5.4). Se invece $d_t < \infty$ allora *Trova-Cammino-Minimo* restituisce $vero$ ed il cammino aumentante P che unisce un nodo $s \in O_x$ al nodo $t \in D_x$ selezionato, insieme alla quantità di flusso θ, definita in (2.14), che deve essere inviata lungo P.

Questo viene fatto dalla procedura *Aumenta-Flusso*, che implementa l'operazione di composizione $\oplus$, in modo simile alla *Aumenta-Flusso* utilizzata per il flusso massimo. Si noti che se $\theta = e_x(s)$ allora il nodo s risulterà bilanciato rispetto al nuovo flusso, ed analogamente per il nodo t se $\theta = -e_x(t)$; altrimenti, θ è determinato dalla capacità del cammino, il che significa che almeno un arco di P diviene saturo oppure vuoto.

Esercizio 2.34 *Si fornisca una descrizione formale, in pseudo-codice, delle procedure* Trova-Cammino-Minimo *e* Aumenta-Flusso*.*

Siccome l'algoritmo usa sempre cammini aumentanti di costo minimo, per il Teorema 2.11 ad ogni passo lo pseudoflusso x è minimale: quindi, se l'algoritmo termina con $g(x) = 0$, allora x è un flusso ottimo. La terminazione dell'algoritmo può essere facilmente provata nel caso in cui b e u siano interi. Infatti, in questo caso lo pseudoflusso iniziale è anch'esso intero, e quindi lo è la quantità θ a quell'iterazione, e quindi lo è anche lo pseudoflusso x ottenuto al termine dell'iterazione. Di conseguenza, ad ogni iterazione x è intero, $\theta \geq 1$ e $g(x)$ diminuisce di almeno un'unità, e quindi l'algoritmo termina in un numero finito di iterazioni. Da questa analisi segue:

Teorema 2.12 *Se le capacità degli archi ed i bilanci dei nodi sono interi, allora per qualsiasi scelta dei costi degli archi esiste almeno una soluzione ottima intera per il problema (MCF).*

Questa *proprietà di integralità* è molto importante per le applicazioni: si pensi ad esempio al caso in cui il flusso su un arco rappresenta il numero di camion, o carrozze ferroviare, o containers, o ancora pacchetti in una rete di comunicazione.

Esempio 2.19:
Sia data l'istanza del problema (MCF) descritta in figura 2.23. Il funzionamento dell'algoritmo basato su cammini minimi successivi è mostrato in figura 2.24.

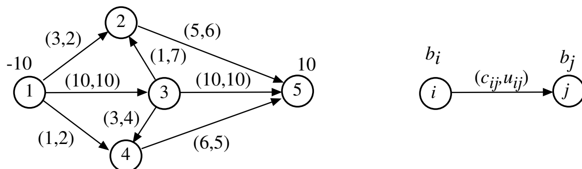

Figura 2.23: Un'istanza per il problema (MCF)

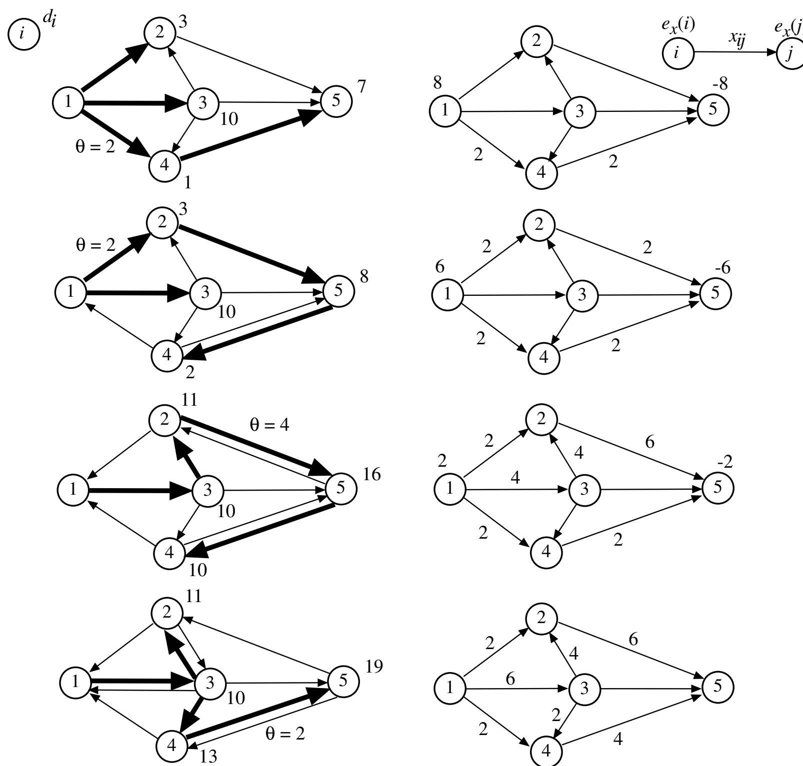

Figura 2.24: Esecuzione dell'algoritmo basato su cammini minimi successivi

Siccome tutti i costi sono non negativi, la procedura *Inizializza* costruisce uno pseudoflusso iniziale identicamente nullo. Per ogni iterazione viene mostrato a sinistra il grafo residuo G_x e a destra lo pseudoflusso ottenuto al termine dell'iterazione. Nel grafo residuo non sono riportati (per chiarezza di visualizzazione) i costi degli archi, ma è evidenziato l'albero dei cammini minimi con i valori delle corrispondenti etichette; è inoltre mostrato il valore θ del flusso inviato lungo il relativo cammino aumentante da 1 a 5. I valori del flusso e degli sbilanciamenti sono mostrati solamente per quegli archi/nodi in cui sono di-

versi da zero. Al termine della quarta iterazione tutti i nodi hanno sbilanciamento nullo, e quindi la soluzione è ottima.

Analizziamo la complessità dell'algoritmo. Lo sbilanciamento complessivo dello pseudoflusso x costruito da *Inizializza* è limitato superiormente da $\overline{g} = \sum_{c_{ij}<0} u_{ij} + \sum_{b_i>0} b_i$; siccome $g(x)$ diminuisce di almeno un'unità ad ogni iterazione, il numero di iterazioni non potrà eccedere $\overline{g}$. È facile vedere che tutte le operazioni effettuate durante una singola iterazione hanno complessità al più $O(n)$, esclusa l'invocazione della procedura *Trova-Cammino-Minimo*: se utilizziamo l'algoritmo $SPT.L$ in cui Q è implementato come $fila$, che ha complessità $O(mn)$, la procedura *Cammini-Minimi-Successivi* risulta avere la complessità pseudopolinomiale $O(\overline{g}mn)$.

2.6.3 Algoritmo basato su cancellazione di cicli

Il Lemma 2.7 suggerisce anche un diverso approccio per la soluzione del problema di flusso di costo minimo: si determina un flusso ammissibile, e poi si utilizzano cicli aumentanti di costo negativo per ottenere flussi ammissibili di costo inferiore. L'algoritmo termina quando non esistono più cicli aumentanti di costo negativo: il flusso ammissibile così determinato è sicuramente di costo minimo.

Quando si satura un ciclo aumentante C, facendo circolare lungo i suoi archi un flusso pari a $\theta(C, x)$, si dice che si "cancella" il ciclo C in quanto esso non risulta più aumentante per il flusso $x(\theta)$ (non si può però escludere che C possa tornare ad essere aumentante per flussi generati successivamente). Il seguente algoritmo, *Cancella-Cicli*, è basato sulla "cancellazione" dei cicli aumentanti (semplici) di costo negativo.

La procedura *Flusso-Ammissibile* determina, se esiste, un flusso ammissibile: in tal caso restituisce *vero* ed il flusso x, altrimenti restituisce $falso$. Una possibile implementazione di questa procedura è stata discussa nel paragrafo 2.5.4.

La procedura *Trova-Ciclo* determina, dato x, se esiste un ciclo aumentante rispetto ad x di costo negativo: in questo caso restituisce *vero* ed il ciclo individuato C, con il suo verso e la sua capacità $\theta = \theta(C, x)$, altrimenti restituisce $falso$. Dal Lemma 2.6 segue che il problema di determinare un ciclo aumentante di costo negativo in G rispetto ad x è equivalente al problema di determinare un ciclo orientato e di costo negativo in G_x; tale problema può essere risolto in diversi modi, ad esempio utilizzando la procedura $SPT.L$ in cui Q è implementato come una fila. In particolare, si può utilizzare la procedura per il problema dell'albero dei cammini minimi con radici multiple (si veda il paragrafo 2.3.7) con insieme di radici $R = N$; ciò corrisponde ad aggiungere al grafo residuo una radice fittizia r e un arco (r, j) di costo $c_{rj} = 0$ per ogni nodo $j \in N$.

Una volta determinato il ciclo C ed il valore θ, la procedura *Cambia-Flusso* costruisce il nuovo flusso $x(\theta) = x \oplus \theta C$.

Procedure *Cancella-Cicli*$(G, c, b, u, x, caso)$:
  begin
    if *Flusso-Ammissibile*(G, c, b, u, x) **then**
      begin
        while *Trova-Ciclo*(G, c, u, x, C, θ) **do** *Cambia-Flusso*(x, C, θ);
        caso := "ottimo"
      end
      else *caso* := "vuoto"
  end.

Procedura 2.10: *Algoritmo basato sulla cancellazione di cicli*

Esercizio 2.35 *Si fornisca una descrizione formale, in pseudo-codice, delle procedure* Trova-Ciclo *e* Cambia-Flusso*.*

Si noti come l'algoritmo basato sulla cancellazione di cicli sia un chiaro esempio di algoritmo di ricerca locale, in cui l'intorno di x è dato da tutti i flussi ottenibili da x inviando flusso lungo un ciclo aumentante semplice.

Esempio 2.20:
Consideriamo di nuovo l'istanza del problema (MCF) descritta in figura 2.23. Il funzionamento dell'algoritmo basato sulla cancellazione di cicli è mostrato in figura 2.25.

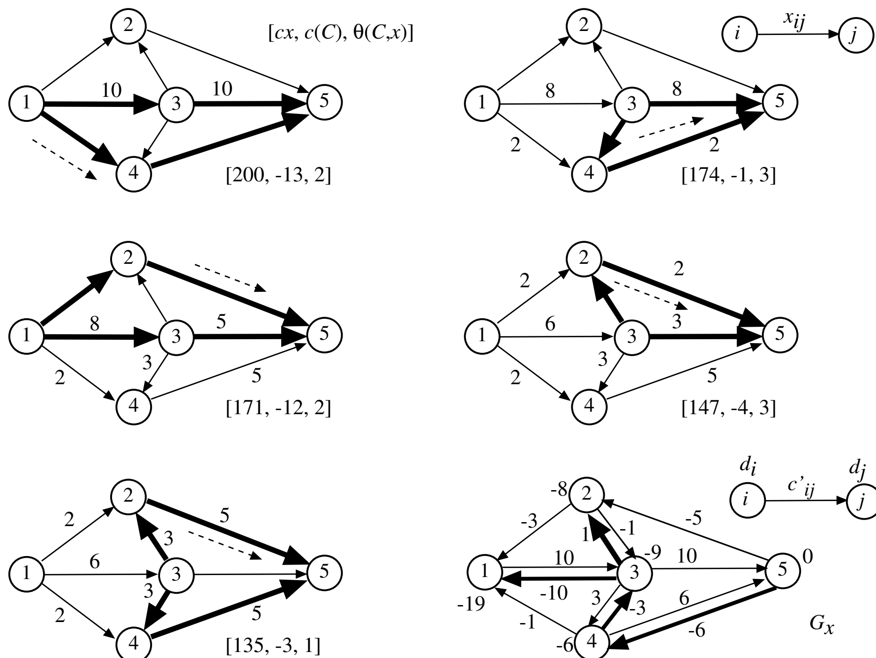

Figura 2.25: Esecuzione dell'algoritmo *Cancella-Cicli*

Il grafo in alto a sinistra descrive l'istanza del problema; una soluzione ammissibile può essere facilmente ottenuta inviando 10 unità di flusso da 1 a 5 lungo gli archi $(1, 3)$ e

(3, 5). I successivi cinque grafi (da sinistra a destra, dall'alto in basso) mostrano i passi svolti dall'algoritmo a partire da tale soluzione ammissibile: per ogni iterazione vengono mostrati il valore del flusso su tutti gli archi in cui non è nullo, il ciclo selezionato (in grassetto) ed il suo verso (freccia tratteggiata), il valore della funzione obiettivo, $c(C)$ e $\theta(C,x)$. Il grafo in basso a destra fornisce la *dimostrazione* che la soluzione determinata è effettivamente ottima: in figura viene mostrato il grafo residuo G_x corrispondente a tale soluzione, con i relativi costi degli archi, ed il corrispondente albero dei cammini minimi con insieme di radici $R = N$, con le relative etichette (è facile verificare che le condizioni di Bellman sono rispettate). Dato che G_x ammette un albero dei cammini minimi, non esistono cicli orientati di costo negativo su G_x, e quindi non esistono cicli aumentanti rispetto ad x di costo negativo, il che garantisce che x sia un flusso ottimo.

La correttezza dell'algoritmo *Cancella-Cicli* discende direttamente dal Lemma 2.7; analizziamo adesso la sua terminazione.

Innanzitutto notiamo che, se le capacità degli archi ed i bilanci dei nodi sono numeri interi, allora ad ogni passo dell'algoritmo il flusso x è intero: infatti possiamo assumere che il flusso restituito da *Flusso-Ammissibile* sia intero (si veda il Teorema 2.8), e replicare gli argomenti già usati per l'algoritmo dei cammini minimi successivi.

Sia $\bar{u} = \max\{u_{ij} : (i,j) \in A\}$ la massima capacità degli archi, che assumiamo finita, e sia $\bar{c} = \max\{|c_{ij}| : (i,j) \in A\}$ il massimo valore assoluto dei costi degli archi: è facile verificare che il costo di qualunque soluzione ammissibile è compreso tra $m\bar{u}\bar{c}$ e $-m\bar{u}\bar{c}$. Se tutti i costi sono interi, il costo di qualsiasi ciclo aumentante utilizzato dall'algoritmo sarà sicuramente inferiore od uguale a -1; siccome $\theta \geq 1$, ad ogni iterazione il valore della funzione obiettivo diminuisce di almeno un'unità, e quindi non possono essere fatte più di $O(m\bar{u}\bar{c})$ iterazioni. Quindi, se i vettori b, c ed u hanno tutte componenti finite e intere, allora l'algoritmo termina.

La complessità dell'algoritmo basato sulla cancellazione di cicli dipende dal modo in cui è realizzata la procedura *Trova-Ciclo*. Poiché la procedura *SPT.L* in cui Q è implementato come *fila* permette di determinare un ciclo di costo negativo in $O(mn)$, l'algoritmo *Cancella-Cicli* ha complessità pseudopolinomiale $O(nm^2\bar{u}\bar{c})$.

2.6.4 Basi di cicli

Nel paragrafo precedente abbiamo descritto un'implementazione della procedura *Trova-Ciclo* basata sulla ricerca, mediante *SPT.L*, di cicli negativi sul grafo residuo G_x; esistono molte altre varianti dell'algoritmo basato sulla cancellazione di cicli, aventi complessità ed efficienza computazionale in pratica molto diverse, che si differenziano proprio per l'implementazione di *Trova-Ciclo*.

Un'altra tecnica per determinare efficacemente cicli negativi è quella basata sulle *basi di cicli*. Abbiamo già notato che un qualsiasi ciclo può essere ottenuto mediante composizione di cicli semplici; diciamo che un insieme di cicli semplici (senza aver fissato un verso di percorrenza) è una base di cicli

se ogni ciclo può essere ottenuto mediante composizione di cicli dell'insieme, scegliendo opportunamente il loro verso di percorrenza. Una base di cicli è facilmente ottenibile mediante un albero di copertura $T = (N, A_T)$ del grafo G: ogni arco $(i, j) \in A \setminus A_T$ induce su T un ciclo $C_T(i, j)$ (si veda l'Appendice B), e l'insieme

$$B(T) = \{C_T(i, j) : (i, j) \in A \setminus A_T\}$$

è la base di cicli indotta da T. Una base di cicli ha pertanto cardinalità $|B(T)| = m - n + 1$.

La proprietà fondamentale delle basi di cicli è descritta dal seguente teorema, di cui omettiamo la dimostrazione.

Teorema 2.13 *Data una base di cicli $B(T)$, se esiste un ciclo C di costo negativo, allora esiste un ciclo $C' \in B(T)$ di costo negativo.*

Questa proprietà permette di costruire algoritmi che utilizzano le basi di cicli per determinare l'esistenza di un ciclo ammissibile di costo negativo. Ad ogni iterazione si ha un albero di copertura T ed un flusso ammissibile x con la proprietà che tutti gli archi $(i, j) \notin T$ sono o saturi ($x_{ij} = u_{ij}$) oppure vuoti ($x_{ij} = 0$): di conseguenza, ogni ciclo $C_T(i, j) \in B(T)$ può essere aumentante solo in uno dei due versi (concorde con (i, j) se l'arco è vuoto, discorde con (i, j) se l'arco è saturo). Con questi elementi, si può dimostrare che se nessuno dei cicli di $B(T)$, scelti con l'unico verso che può renderli aumentanti, ha costo negativo, allora non esistono cicli aumentanti di costo negativo e l'algoritmo termina dichiarando che x è ottimo. Altrimenti, viene scelto un ciclo di costo negativo $C_T(i, j) \in B(T)$ (col verso opportuno) e viene aggiornato il flusso lungo di esso. L'aggiornamento rende vuoto o saturo uno degli archi di $C_T(i, j)$ in $B(T)$: tale arco viene quindi eliminato da T e sostituito con (i, j), in modo da considerare nell'iterazione successiva una diversa base di cicli. Si noti che può accadere $\theta(C_T(i, j), x) = 0$, ossia che uno degli archi discordi del ciclo è già vuoto oppure uno degli archi concordi è già saturo: in questo caso il flusso non cambia e cambia solamente la base $B(T)$. Ad ogni iterazione, quindi, ci si sposta da una base $B(T)$ ad una base $B(T')$ "adiacente", ossia tale che T' è ottenuto da T mediante lo scambio di due archi. È possibile dimostrare che, dopo un numero finito di iterazioni, si ottiene un albero di copertura T tale che tutti i cicli $C_T(i, j) \in B(T)$ (col verso opportuno) hanno costo non negativo, e quindi il flusso x è ottimo.

Gli algoritmi basati sulle basi di cicli sono conosciuti come "Algoritmi del Simplesso su Reti". Nella letteratura scientifica sono stati proposti moltissimi algoritmi per il flusso di costo minimo: alcuni di questi algoritmi possono essere considerati versioni (molto) raffinate di quelli illustrati in questo corso, mentre altri algoritmi sono basati su idee completamente diverse. Per maggiori dettagli su questi temi si rinvia alla letteratura indicata ed a corsi successivi dell'area di Ricerca Operativa.

2.7 Problemi di accoppiamento

Sia $G = (O \cup D, A)$ un grafo bipartito non orientato, dove $O = \{1, \dots, n\}$ è l'insieme dei *nodi origine*, $D = \{n+1, \dots, n+d\}$ è l'insieme dei nodi destinazione, e $A \subseteq O \times D$, con $|A| = m$, è l'insieme degli archi, ai quali possono essere associati costi c_{ij}. Non è restrittivo supporre $n \leq d$.

Un *accoppiamento* (*matching*) M è un sottoinsieme di archi che non hanno nodi in comune. Gli archi in M sono detti *interni*, mentre gli archi in $A \setminus M$ sono detti *esterni*. Dato un accoppiamento M, un nodo i è *esposto* rispetto a M se nessun arco di M incide in i, altrimenti i è detto *accoppiato*; indicheremo con O_M e D_M gli insiemi dei nodi rispettivamente in O e D che sono esposti. Nel caso in cui $|O| = |D|$, cioè $d = n$, M è un *accoppiamento perfetto* (o *assegnamento*) se nessun nodo è esposto, ovvero se $|M| = n$. Un esempio è fornito in figura 2.26.

La *cardinalità* di un accoppiamento M è $|M|$, mentre il costo $C(M)$ di un accoppiamento M è la somma dei costi degli archi di M (si assume $C(\emptyset) = 0$). Dato un accoppiamento $M \neq \emptyset$, l'arco $(i,j) \in M$ di costo massimo è detto *arco bottleneck* (collo di bottiglia) di M; il valore $V(M) = \max\{c_{ij} : (i,j) \in M\}$ è detto il *valore bottleneck* di M.

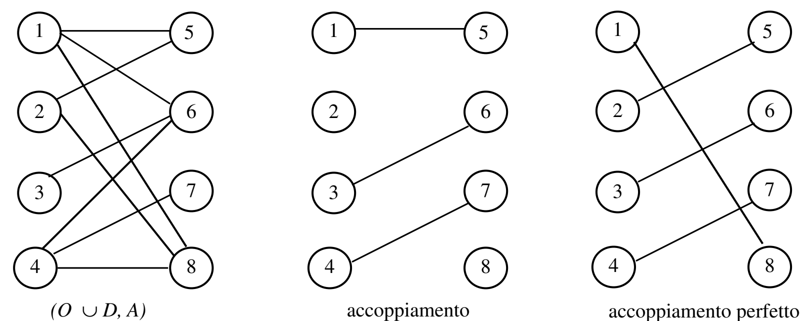

Figura 2.26: Esempi di accoppiamenti

Nel seguito studieremo i seguenti problemi:

1. *Accoppiamento di massima cardinalità*: si vuole determinare un accoppiamento di massima cardinalità in G.

2. *Assegnamento di costo minimo*: si vuole determinare, tra tutti gli accoppiamenti perfetti in G, uno che abbia di costo minimo.

3. *Assegnamento di massima cardinalità bottleneck*: si vuole determinare, tra tutti gli accoppiamenti di massima cardinalità in G, uno che abbia valore bottleneck minimo, cioè tale che il massimo costo degli archi sia minimo.

2.7.1 Accoppiamento di massima cardinalità

Il problema di accoppiamento di massima cardinalità in un grafo $G = (O \cup D, A)$ può essere trasformato in un problema equivalente di flusso massimo con più sorgenti e pozzi sul grafo orientato $\bar{G} = (N, \bar{A})$ dove $\bar{A}$ contiene gli archi di A orientati dai nodi in O a quelli in D. Ogni arco $(i, j) \in \bar{A}$ ha capacità superiore $u_{ij} = 1$, O è l'insieme delle sorgenti, D è l'insieme dei pozzi, ed ogni sorgente/pozzo può immettere nella/prelevare dalla rete un'unità di flusso. Equivalentemente, si possono aggiungere a $\bar{G}$ una "super sorgente" s ed un "super pozzo" t, collegati rispettivamente a tutti i nodi di O e D da archi di capacità unitaria, e considerare il problema di flusso massimo da s a t.

È facile verificare che l'insieme degli archi saturi in qualunque flusso ammissibile (intero) x in $\bar{G}$ forma un accoppiamento M in G la cui cardinalità è pari al valore v del flusso; viceversa, da un qualunque accoppiamento M si costruisce un flusso ammissibile. Nell'esempio in figura 2.27, relativo al grafo G di figura 2.26, è mostrato in (a) un accoppiamento M con $|M| = 3$, ed in (b) il corrispondente flusso ammissibile x su $\bar{G}$ con valore del flusso $v = 3$ (sono indicati solo i flussi diversi da zero).

È quindi possibile risolvere il problema dell'accoppiamento di massima cardinalità in G applicando un qualsiasi algoritmo per il problema del flusso massimo (con più sorgenti e pozzi) in $\bar{G}$. Data la particolare struttura del problema, però, alcune operazioni degli algoritmi possono essere implementate in maniera più efficiente, o hanno un particolare significato che è possibile sfruttare ai fini algoritmici.

Si consideri ad esempio il concetto di cammino aumentante sul grafo $\bar{G}$ rispetto ad un qualche flusso x che rappresenta un accoppiamento M, ossia tale che $x_{ij} = 1$ per $(i, j) \in \bar{A}$ se e solo se $(i, j) \in M$. Un arco $(i, j) \in \bar{A}$ è saturo se e solo se il corrispondente arco $(i, j) \in A$ è interno ad M. Siccome il grafo è bipartito, i nodi di qualsiasi cammino su $\bar{G}$ devono appartenere alternativamente ad O ed a D. Ma tutti gli archi $(i, j) \notin M$ su G corrispondono ad archi vuoti su $\bar{G}$: quindi, tali archi possono essere utilizzati in un cammino aumentante solamente in modo concorde col loro verso, ossia da un nodo di O ad un nodo di D. Viceversa, tutti gli archi $(i, j) \in M$ su G corrispondono ad archi saturi su $\bar{G}$: quindi, tali archi possono essere utilizzati in un cammino aumentante solamente in modo discorde al loro verso, ossia da un nodo di D ad un nodo di O. Da tutto questo segue che un cammino aumentante su $\bar{G}$ rispetto ad un flusso x corrisponde ad un *cammino alternante* su G rispetto all'accoppiamento M, ossia un cammino formato alternativamente da archi esterni ed archi interni rispetto a M.

Non tutti i cammini alternanti su G rappresentano però cammini aumentanti su $\bar{G}$; affinché questo accada, occorre anche che il cammino parta da un'origine esposta e termini in una destinazione esposta (in questo caso, il cammino alternante è detto *aumentante*). Infatti, le origini/destinazioni

esposte sono quelle per cui non transita ancora nessun flusso: siccome ogni origine/destinazione ha "capacità" unitaria, le origini esposte sono i nodi nell'insieme S_x delle sorgenti "attive" e le destinazioni esposte sono i nodi nell'insieme T_x dei pozzi "attivi" (si veda il paragrafo 2.5.4). Per questo, qualsiasi cammino aumentante su $\bar{G}$ deve avere come primo nodo un'origine esposta e come ultimo nodo una destinazione esposta. Esiste quindi una corrispondenza biunivoca tra cammini aumentanti su $\bar{G}$ e *cammini alternanti aumentanti* su G. Un cammino alternante P su G è aumentante se, detti $P_E = P \setminus M$ e $P_I = M \cap P$ l'insieme degli archi esterni e quello degli archi interni di P, si ha $|P_E| - |P_I| = 1$, ossia gli archi esterni sono esattamente uno in più di quelli interni.

Esempio 2.21:
Le affermazioni precedenti possono essere facilmente verificate nell'esempio in figura 2.27; in (c) e (d) sono mostrati rispettivamente un cammino alternante aumentante su G rispetto all'accoppiamento M ed un cammino aumentante su $\bar{G}_x$ rispetto al flusso x.

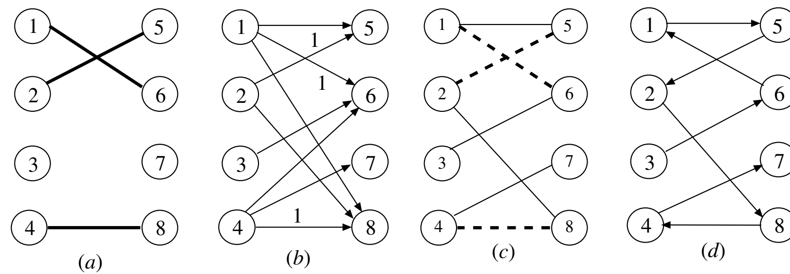

Figura 2.27: Flussi, accoppiamenti e cammini (alternanti) aumentanti

Si noti che, dato un cammino (alternante) aumentante, la capacità del cammino è sempre 1. Questo corrisponde al fatto che, in questo caso, l'operazione di composizione $x' := x \oplus P'$, dove P' è un cammino aumentante su $\bar{G}$, corrisponde a

$$M' := M \oplus P = M \setminus P_I \cup P_E$$

dove P è un cammino alternante aumentante su G: in altre parole, l'operazione di composizione corrisponde a togliere da M gli archi interni di P ed aggiungere quelli esterni di P. Siccome $|P_E| = |P_I| + 1$, si ha che $|M \oplus P| = |M| + 1$; infatti, il nuovo flusso x' ha valore $v' = v + 1$.

Esempio 2.22:
Proseguendo l'esempio di figura 2.27, e applicando l'operazione di composizione aull'accoppiamento M mostrato in (a) e al cammino P mostrato in (c), si ottiene il nuovo accoppiamento $M' = \{(1,5),(2,8),(3,6),(4,7)\}$. È immediato verificate che il nuovo accoppiamento corrisponde al flusso che si ottiene dal flusso mostrato in (b) inviando un'unità di flusso lungo il cammino aumentante mostrato in (d).

Con queste notazioni, possiamo costruire una versione specializzata dell'algoritmo *Cammini-Aumentanti* per risolvere il problema dell'accoppiamento di massima cardinalità. L'inizializzazione, $M := \emptyset$, corrisponde a scegliere $x = 0$ come flusso iniziale.

La procedura *Cammino-Aumentante* determina, se esiste, un cammino alternante aumentante: per questo è sufficiente visitare il grafo bipartito G partendo dai nodi di O_M e visitando alternativamente archi esterni e interni, il che corrisponde alla procedura *Visita* con semplici modifiche. Si noti che, rispetto al caso del flusso massimo, il controllo di ammissibilità di un arco è più semplice, e non è necessario determinare la capacità del cammino. Se alla fine della visita non si è raggiunto alcun nodo in D_M allora non esistono cammini aumentanti e l'accoppiamento è di massima cardinalità: ciò corrisponde al fatto che non esistono cammini aumentanti su $\bar{G}_x$, e quindi il flusso x ha valore massimo. In particolare, questo accade sicuramente qualora $O_M = \emptyset$, ossia se è già stato prodotto un accoppiamento di massima cardinalità.

```
Procedure Accoppiamento-MaxCard(O, D, A, M):
  begin
    M := ∅;
    while Cammino-Aumentante(O, D, A, M, P) do
      Cambia-Accoppiamento(M, P)
  end.
```

Procedura 2.11: Algoritmo *Accoppiamento-MaxCard*

Se invece viene determinato un cammino alternante aumentante P, viene invocata la procedura *Cambia-Accoppiamento* che realizza l'operazione di composizione $M \oplus P$; tale operazione è analoga alla *Aumenta-Flusso* dell'algoritmo *Cammini-Aumentanti*, anche se più semplice.

Esercizio 2.36 *Si fornisca una descrizione formale, in pseudo-codice, delle procedure* Cammino-Aumentante *e* Cambia-Accoppiamento.

La complessità di *Accoppiamento-MaxCard* è $O(mn)$, in qualunque modo venga implementata la visita: infatti, la complessità della generica procedura *Cammini-Aumentanti* è $O(mnU)$, ma in questo caso $U = 1$. In effetti, è immediato verificare che la procedura termina dopo al più n iterazioni, ognuna delle quali richiede una visita del grafo e quindi costa $O(m)$.

Esercizio 2.37 *Si consideri il grafo in figura 2.28; si applichi la procedura* Accoppiamento-MaxCard *fornendo ad ogni iterazione l'accoppiamento, l'albero della visita ed il cammino aumentante.*

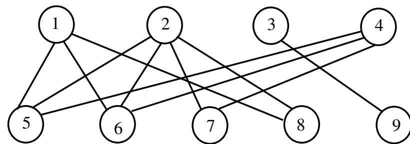

Figura 2.28:

2.7.2 Assegnamento di costo minimo

Analogamente al problema dell'accoppiamento di massima cardinalità, il problema dell'assegnamento costo minimo è equivalente al problema di flusso costo minimo sul grafo orientato $\bar{G}$ in cui le capacità degli archi sono unitarie, i costi degli archi sono quelli del problema di accoppiamento, ogni nodo in O produce un'unità di flusso ed ogni nodo in D consuma un'unità di flusso. La trasformazione è illustrata in figura 2.29 (b) per l'istanza in (a) (per chiarezza di visualizzazione non sono indicate le capacità degli archi, tutte pari a 1).

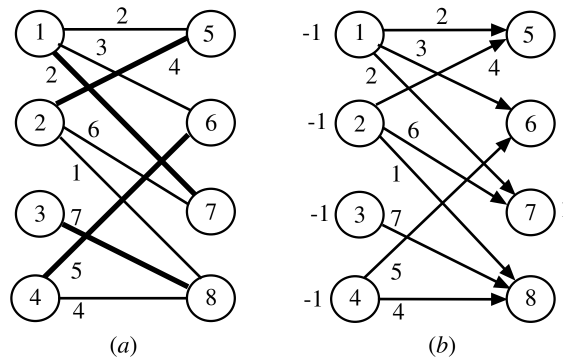

Figura 2.29: Trasformazione in un problema di flusso di costo minimo

Analogamente al caso del problema dell'accoppiamento di massima cardinalità, è possibile specializzare gli algoritmi per il problema del flusso di costo minimo al caso particolare del problema dell'assegnamento di costo minimo. Nell'algoritmo basato sui cammini minimi successivi, ad esempio, si determina ad ogni passo un cammino aumentante di costo minimo che connette un nodo con eccesso di flusso ad uno con difetto di flusso: è immediato verificare che, nel caso del problema dell'assegnamento di costo minimo, ciò corrisponde a determinare il *cammino alternante aumentante di costo minimo* rispetto all'accoppiamento corrente M. Per fare questo si può utilizzare il grafo ausiliario $G_M = (N, A_M)$, tale che

$$A_M = \{(j,i) : (i,j) \in M\} \cup \{(i,j) : (i,j) \in A \backslash M\} ,$$

ove agli archi di G_M sono associati i costi

$$c'_{ij} = \begin{cases} c_{ij}, & \text{se } (i,j) \in A \backslash M, \\ -c_{ji}, & \text{se } (j,i) \in M, \end{cases}$$

È facile verificare che G_M è il grafo residuo $\bar{G}_x$ per lo pseudoflusso x corrispondente all'accoppiamento M. Un qualunque cammino orientato P_{st} da un nodo $s \in O_M$ ad un nodo $t \in D_M$ corrisponde ad un cammino

alternante aumentante P, e viceversa; inoltre, per costruzione il costo di P_{st} in G_M, $C'(P_{st})$, è uguale al costo di P, definito come

$$C(P) = \sum_{(i,j)\in P_E} c_{ij} - \sum_{(i,j)\in P_I} c_{ij} \ .$$

Inviare un'unità di flusso lungo un cammino aumentante corrisponde ad applicare l'operazione di composizione $M' = M \oplus P$ del paragrafo precedente; è facile verificare che risulta $C(M') = C(M) + C(P)$.

Esempio 2.23:
Consideriamo il grafo G e l'accoppiamento M in figura 2.30(a) di costo $C(M) = 3$. Il grafo ausiliario G_M è descritto in figura 2.30(b). Al cammino orientato $P_{37} = \{(3,8), (8,2), (2,7)\}$ nel grafo ausiliario in figura 2.30(b), avente costo 12, corrisponde nel grafo originario il cammino aumentante $P = \{(3,8), (2,8), (2,7)\}$, avente anch'esso costo 12.

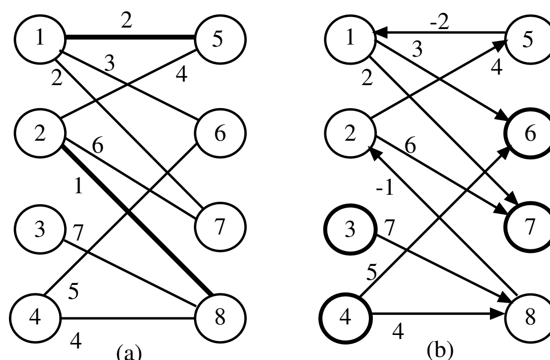

Figura 2.30: Cammini aumentanti e cammini sul grafo ausiliario

Quanto detto finora porta alla definizione del seguente algoritmo per la soluzione del problema.

Procedure *Assegnamento-MinCost*(O, D, A, c, M):
begin
$M = \emptyset$;
while *Cammino-AA-Minimo*(G_M, P) **do**
Cambia-Accoppiamento(M, P, O_M, D_M, G_M)
end.

Procedura 2.12: Algoritmo *Assegnamento-MinCost*

L'algoritmo parte dall'accoppiamento iniziale vuoto, corrispondente al flusso iniziale $x = 0$. Si noti che non è necessario eseguire la procedura *Inizializza* del paragrafo 2.6.2, anche in presenza di costi negativi, in quanto x è sicuramente minimale: il grafo residuo, essendo bipartito, non contiene nessun ciclo orientato.

La procedura *Cammino-AA-Minimo* cerca di determinare un cammino alternante aumentante di costo minimo tra un nodo $s \in O_M$ ed un nodo $t \in D_M$, restituendo *vero* se ha successo e $falso$ altrimenti, nel qual caso l'algoritmo termina. In particolare, restituisce $falso$ se $O_M = \emptyset$, ossia se non esistono nodi esposti in O; altrimenti determina un albero dei cammini minimi con insieme di nodi radice O_M per G_M, col relativo vettore di etichette d e seleziona un nodo $t \in D_M$ (ad esempio, quello con etichetta d_t minore): se $d_t = \infty$ allora t non è connesso ad alcun nodo di O_M e la procedura restituisce $falso$, altrimenti si è determinato il cammino desiderato.

La procedura *Cambia-Accoppiamento* esegue l'operazione di composizione tra l'accoppiamento M corrente ed il cammino P, ed aggiorna gli insiemi dei nodi esposti, O_M e D_M, ed il grafo ausiliario G_M associato al nuovo accoppiamento.

Esercizio 2.38 *Si fornisca una descrizione formale, in pseudo-codice, delle procedure* Cammino-AA-Minimo *e* Cambia-Accoppiamento.

Quando l'algoritmo termina, se $|M| = |O| = |D|$ allora si è determinato un assegnamento di costo minimo, altrimenti non esistono accoppiamenti perfetti in G.

Esercizio 2.39 *Dimostrare l'affermazione precedente.*

Esempio 2.24:
In figura 2.31 sono mostrate due iterazioni dell'algoritmo.

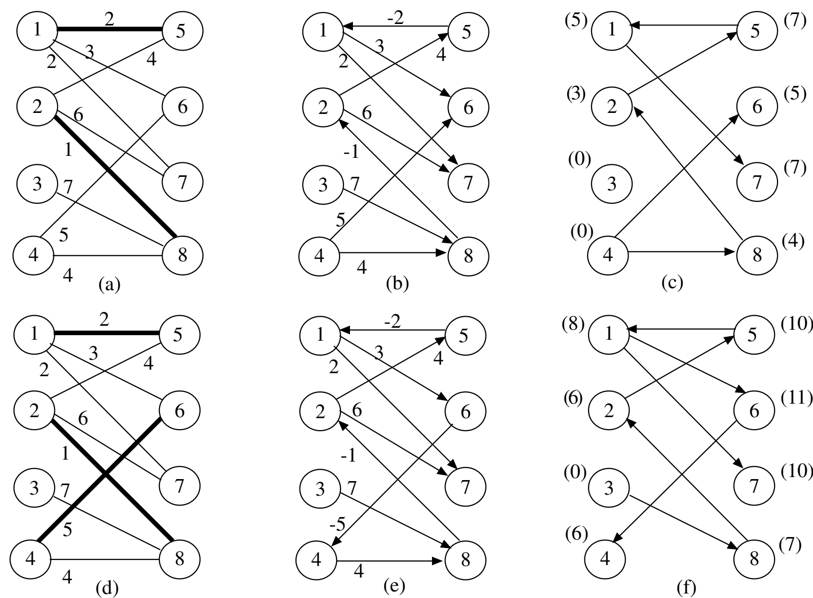

Figura 2.31:

In (a) l'accoppiamento M corrente, in (b) il corrispondente grafo ausiliario G_M ed in (c) l'albero dei cammini minimi con insieme di radici $R = O_M = \{3, 4\}$, con le relative etichette ottime ai nodi. Essendo $D_M = \{6, 7\}$, si pone $t = 6$ poiché $d(6) = 5 < 7 = d(7)$, selezionando così il cammino alternante aumentante $P = \{(4, 6)\}$. In figura 2.31(d) è mostrato il nuovo accoppiamento $M' = M \oplus P = \{(1, 5), (2, 8), (4, 6)\}$, di cardinalità 3 e costo $C(M') = C(M)+C(P) = 3+5 = 8$, mentre in 2.31(e) ed (f) sono mostrati rispettivamente il corrispondente grafo ausiliario $G_{M'}$ e l'albero dei cammini minimi. Il cammino alternante aumentante di costo minimo è $P' = \{(3, 8), (2, 8), (2, 5), (1, 5), (1, 7)\}$, con $C(P') = d(7) = 10$. L'assegnamento ottimo, $M'' = M' \oplus P' = \{(1, 5), (2, 8), (4, 6)\} \setminus \{(2, 8), (1, 5)\} \cup \{(3, 8), (2, 5), (1, 7)\} = \{(1, 7), (2, 5), (3, 8), (4, 6)\}$, di costo $C(M'') = C(M') + c(P') = 8 + 10 = 18$, è mostrato in figura 2.29(a).

La correttezza dell'algoritmo deriva direttamente dalla correttezza della procedura *Cammini-Minimi-Successivi* per il problema del flusso di costo minimo; dall'analisi svolta per quella procedura risulta immediatamente che, se utilizziamo l'algoritmo *SPT.L* in cui Q è implementato come *fila* per implementare *Cammino-AA-Minimo*, la complessità di *Assegnamento-MinCost* è $O(mn^2)$, essendo n il massimo numero di iterazioni.

È possibile dimostrare che l'algoritmo *Assegnamento-MinCost* può essere usato anche per risolvere un problema più generale dell'assegnamento di costo minimo, ossia il problema dell'*accoppiamento di massima cardinalità e costo minimo*, nel quale si vuole determinare, tra tutti gli accoppiamenti di massima cardinalità (anche se non necessariamente perfetti), quello di costo minimo. Per fare questo è solamente necessario garantire che, ad ogni iterazione, il nodo $t \in D_M$ selezionato come destinazione sia uno di quelli di etichetta minima, ossia $d_t = \min\{d_j : j \in D_M\}$, ovvero che il cammino alternante aumentante selezionato sia di costo minimo tra *tutti* i cammini che uniscono *un qualsiasi* nodo di O_M ad un nodo di D_M.

2.7.3 Accoppiamento di massima cardinalità bottleneck

Il problema dell'accoppiamento di massima cardinalità bottleneck non è facilmente formulabile come problema di flusso di costo minimo, ma può essere facilmente risolto utilizzando come sottoprogrammi algoritmi visti nei paragrafi precedenti. Descriveremo le idee di base per il caso, più semplice, dell'assegnamento bottleneck, estendendole in seguito al caso più generale.

Si supponga di conoscere il valore bottleneck ottimo

$$z = \min\{V(M) : M \text{ è un assegnamento in } G\} ,$$

e si consideri il grafo parziale $G_v = (O, D, A_v)$, parametrico rispetto al valore reale v, dove

$$A_v = \{(i, j) \in A : c_{ij} \leq v\} .$$

Dalla definizione discende che se $v < z$, allora G_v non contiene accoppiamenti perfetti, altrimenti $(v \geq z)$ G_v contiene almeno un assegnamento. Ciò suggerisce il seguente algoritmo:

1. si parte da un valore di v abbastanza piccolo, ossia tale che sicuramente $v \leq z$ (ad esempio $v = \min\{c_{ij} : (i,j) \in A\}$);

2. si calcola un accoppiamento M_v di massima cardinalità in G_v: se $|M_v| = n$ ci si ferma, altrimenti si determina il più piccolo valore di v che permette di aggiungere archi ad A_v (cioè il minimo costo di un arco strettamente maggiore di v) e si itera.

Il processo di aggiunta di archi ad A_v viene iterato fino a che M_v non sia un accoppiamento perfetto, oppure sino a che $A_v = A$ e $|M_v| < n$: nel primo caso M_v è un assegnamento bottleneck, ossia ha valore $V(M_v) = v$ minimo tra tutti i possibili assegnamenti, mentre nel secondo caso il grafo G è privo di accoppiamenti perfetti. In questo caso, operando opportunamente, si può garantire che l'ultimo accoppiamento M_v prodotto sia bottleneck tra tutti gli accoppiamenti di massima cardinalità.

Procedure *Accoppiamento-Bottleneck*(O, D, A, c, M_v, v):
  begin
    Inizializza(v, A_v); $M_v := \emptyset$;
    repeat
      Accoppiamento-MaxCard(O, D, A_v, M_v);
      if $|M_v| < n$ **then**
        begin
          $v := \min\{c_{ij} : (i,j) \notin A_v\}$;
          $A_v := A_v \cup \{(i,j) \notin A_v : c_{ij} = v\}$;
        end
    until $A_v = A$ **or** $|M_v| = n$ **do**
  end.

Procedura 2.13: Algoritmo *Accoppiamento-Bottleneck*

La procedura *Inizializza* determina un opportuno valore v ed il corrispondente insieme di archi A_v. Se si vuole trovare un assegnamento è possibile scegliere

$$v = \max\{\min\{c_{ij} : (i,j) \in S(i)\} : i \in O \cup D\}$$

poiché per valori inferiori almeno un nodo risulterebbe isolato dagli altri, impedendo l'esistenza di un accoppiamento perfetto in G_v. Se invece si vuole risolvere il problema dell'accoppiamento bottleneck di massima cardinalità è possibile scegliere $v = \min\{c_{ij} : (i,j) \in A\}$, poiché per valori inferiori A_v è vuoto.

La procedura *Accoppiamento-MaxCard* determina un accoppiamento di massima cardinalità in G_v. Si noti che, durante il processo iterativo, è possibile utilizzare come accoppiamento di partenza l'accoppiamento di massima cardinalità determinato all'iterazione precedente, che è sicuramente

ancora un accoppiamento valido in quanto tutti gli archi che erano in A_v all'iterazione precedente ci sono anche in quella attuale (v è crescente).

Poiché ad ogni iterazione si aggiunge ad A_v almeno un arco, non si effettueranno più di m iterazioni. Se si modifica la procedura *Accoppiamento-MaxCard* in modo da utilizzare come accoppiamento di partenza l'accoppiamento M_v in input, si determineranno al più n cammini aumentanti durante l'intera esecuzione dell'algoritmo; si può dimostrare quindi che la complessità di tutte le chiamate ad *Accoppiamento-MaxCard* è $O(mn)$. Per determinare in modo efficiente il nuovo valore di v a ciascuna iterazione è sufficiente ordinare A all'inizio in modo tale che, globalmente, il costo di determinare il nuovo valore di v e gli archi da aggiungere ad A_v sia $O(m)$. Siccome l'ordinamento costa $O(m \log n)$ e viene fatto una volta sola, la complessità di *Accoppiamento-Bottleneck* è $O(mn)$.

Esempio 2.25:
Si vuole determinare un assegnamento bottleneck nel grafo G di figura 2.32(a). Il valore di partenza è $v = 4$, corrispondente al minimo costo degli archi uscenti dal nodo 4, poiché per valori inferiori il nodo 4 risulterebbe isolato dagli nodi. In figura 2.32(b) vengono mostrati il grafo parziale G_4 e l'accoppiamento M_4 con $|M_4| = 3$.

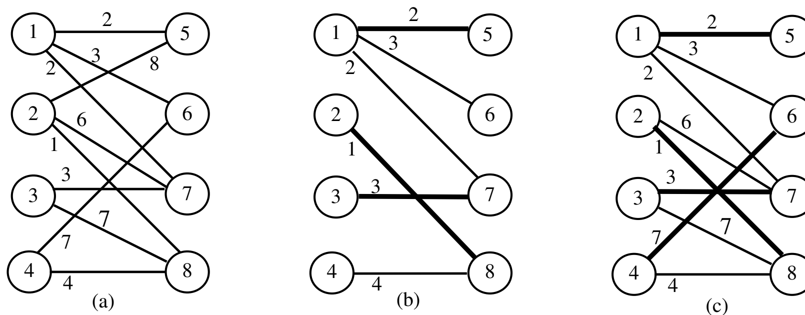

Figura 2.32:

Nella prima iterazione $v = 6$, ma l'aggiunta dell'arco $(2, 7)$ non modifica l'accoppiamento, cioè $M_6 = M_4$. All'iterazione successiva, $v = 7$; in figura 2.32(c) viene mostrato il grafo G_7 ed il nuovo accoppiamento $M_7 = \{(1, 5), (2, 8), (3, 7), (4, 6)\}$ ottenuto da M_4 mediante il cammino aumentante $P = \{(4, 6)\}$. M_7 risulta essere perfetto; si tratta quindi di un assegnamento bottleneck, con valore $V(M) = 7$.

Utilizzare la versione modificata di *Accoppiamento-MaxCard* che riparte dall'accoppiamento M_v in input non è soltanto un utile accorgimento che permette di velocizzare l'algoritmo, ma è necessario per la correttezza nel caso in cui non esistano accoppiamenti perfetti. Si consideri ad esempio il caso del grafo $G = (O \cup D, A)$ con $O = \{1, 2\}$, $D = \{3, 4\}$, $A = \{(1, 3), (2, 3)\}$, $c_{13} = 1$ e $c_{23} = 10$. Al primo passo della procedura $v = 1$ e quindi $A_v = \{(1, 3)\}$: la procedura *Accoppiamento-MaxCard* determina $M_v = \{(1, 3)\}$. Al secondo passo $v = 10$ e quindi $A_v = A$. Se

Accoppiamento-MaxCard partisse con l'accoppiamento iniziale $M = \emptyset$, potrebbe determinare come accoppiamento di massima cardinalità su G_v sia $M' = \{(1,3)\}$ che $M'' = \{(2,3)\}$, dato che entrambe sono accoppiamenti di cardinalità 1. Chiaramente solo M' è un accoppiamento bottleneck di massima cardinalità, per cui se venisse determinato M'' l'algoritmo darebbe una risposta errata.

In generale, se non esistono assegnamenti ed il valore bottleneck è minore del massimo costo degli archi, l'algoritmo eseguirà una sequenza finale di iterazioni in cui cerca senza successo di costruire un accoppiamento di cardinalità maggiore di quello disponibile, finchè non esaurisce l'insieme degli archi e termina. Per la correttezza dell'algoritmo, è cruciale che durante queste iterazioni l'algoritmo non modifichi l'accoppiamento corrente costruendo un accoppiamento con la stessa cardinalità ma contenente archi a costo più alto: per garantire questo è sufficiente fare in modo che *Accoppiamento-MaxCard* riparta dal precedente accoppiamento M_v. Infatti, la procedura modificherà l'accoppiamento solo se può aumentarne la cardinalità: quindi, ad ogni iterazione di *Accoppiamento-Bottleneck* l'accoppiamento corrente M_v contiene solo archi il cui costo è minore od uguale del valore v' corrispondente all'ultima iterazione in cui la cardinalità di M_v è aumentata. Questo garantisce che, a terminazione, l'accoppiamento sia bottleneck anche nel caso in cui non sia perfetto.

Riferimenti Bibliografici

R. K. Ahuja, T. L. Magnanti, J. B. Orlin, "**Network flows. Theory, algorithms, and applications**", Prentice Hall, Englewood Cliffs, NJ (1993).

M. S. Bazaraa, J. J. Jarvis, H. D. Sherali, "**Linear programming and network flows**", Wiley, New York, NY (1990).