# 1. Ricerca locale

Talvolta, anche una strategia estremamente "miope" può dare buoni risultati:

"Coltiviamo senza guardare più in là del proprio orticello".

In altri termini, se si conosce una soluzione ammissibile di un problema di ottimizzazione, si può cercare di trovarne una migliore che sia in qualche modo "vicina" a quella precedente. Data una soluzione ammissibile $Sol$ del problema, si definisce un *intorno* $I(Sol)$ di questa soluzione come un insieme di soluzioni ammissibili che sono ottenibili da essa in base a certi criteri di trasformazione. Si può successivamente sostituire la soluzione con una che fa parte del suo intorno ed è "migliore" di essa, e ripetere il procedimento finché non si riesce più a migliorare. Tale strategia è detta ricerca locale e gli algoritmi basati su di essa hanno la seguente struttura generale:

```
ricercaLocale()
    Sol = una soluzione ammissibile del problema
    while esiste S ∈ I(Sol) che è migliore di Sol do
        Sol = S
    return Sol
```

**Esempio (Intorno)** Si consideri il problema del minimo albero di copertura, introdotto nel Capitolo 14. Una soluzione ammissibile $Sol$ del problema è un albero di copertura, un albero avente tutti i nodi di $V$, ma solo $n-1$ degli $m$ archi in $E$. Un possibile intorno è il seguente:

$$I(Sol) = \{T : T \text{ è un albero di copertura ottenuto da } Sol \text{ aggiungendo un arco non in } Sol \text{ e cancellando un altro arco dall'unico circuito che si è formato}\}.$$

Un criterio di miglioramento consiste nello scegliere un albero $T \in I(Sol)$ in cui l'arco aggiunto ha un peso più piccolo dell'arco cancellato. Un esempio è fornito nella Figura 1.1. ■

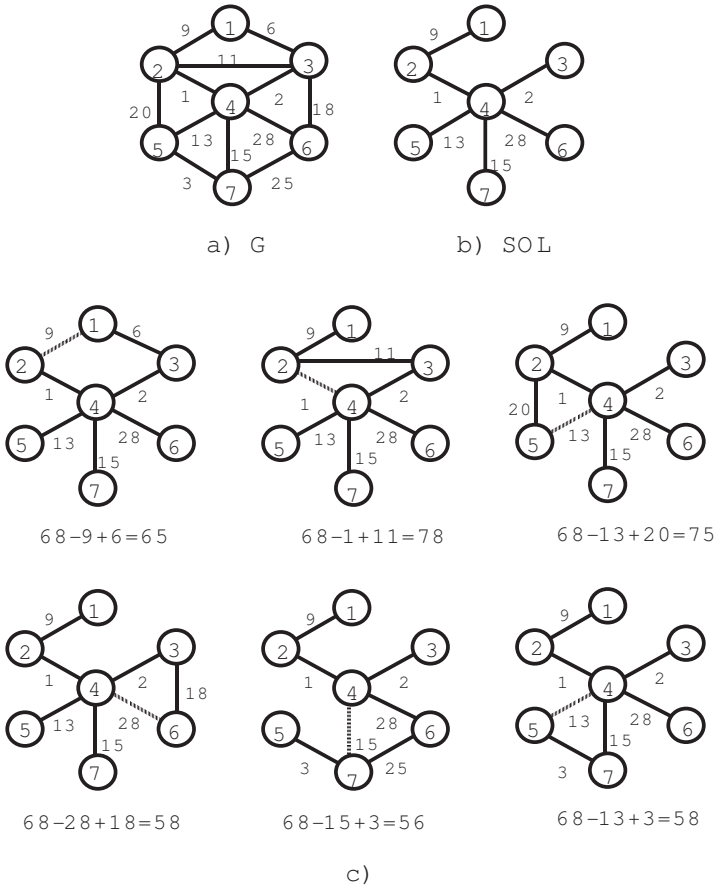


Figura 1.1: Albero di copertura. a) il grafo $G$ della Figura 14.1; b) una soluzione ammissibile *Sol*; c) sei delle tredici soluzioni dell'intorno $I(Sol)$ ottenute aggiungendo un arco e togliendone uno dal circuito formatosi: la migliore si ottiene sostituendo $[4,7]$ con $[5,7]$. **TODO**: "`SOL`" $\rightarrow$ "*Sol* "

In generale, il procedimento di ricerca locale si arresta su una soluzione che è un ottimo locale, cioè che non può essere migliorata attraverso le trasformazioni possibili, ma che può non essere l'ottimo globale del problema. Inoltre, partendo da soluzioni iniziali diverse si possono raggiungere ottimi locali diversi, come illustrato nella Figura 1.2. Ovviamente, se $I(Sol)$ comprende tutte le soluzioni ammissibili del problema, allora ogni ottimo locale è anche ottimo globale, e la ricerca locale dà sempre la soluzione ottima del problema. Una tale ricerca, però, ha senso se la cardinalità di $I(Sol)$ è molto più piccola della cardinalità dell'insieme di tutte le soluzioni ammissibili, e tale restrizione sulla cardinalità di $I(Sol)$ può far arrestare la ricerca su un ottimo locale che non è quello globale!

Perché l'intero procedimento abbia complessità polinomiale, occorre che la ricerca della soluzione $S$ nell'intorno $I(Sol)$ abbia complessità polinomiale e che il numero complessivo di passi di "miglioramento", cioè di ripetizioni del ciclo **while**, sia anch'esso polinomiale. Per alcuni problemi, ma non per tutti, è possibile definire intorni di cardinalità "piccola" per i quali ogni ottimo locale è anche un ottimo globale che inoltre può essere trovato in tempo polinomiale con una ricerca locale.

**Esempio (Minimo albero di copertura)** È possibile dimostrare che l'intorno definito nell'Esempio 1 per il minimo albero di copertura è tale che ogni minimo locale è anche globale.

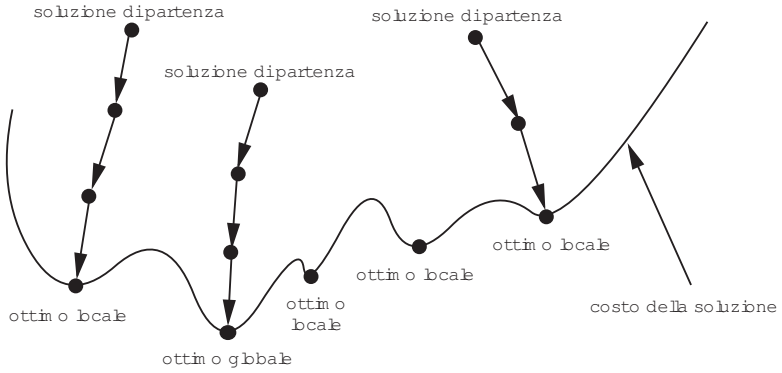


Figura 1.2: Ottimi locali e globali. **TODO**: Cambiare font?

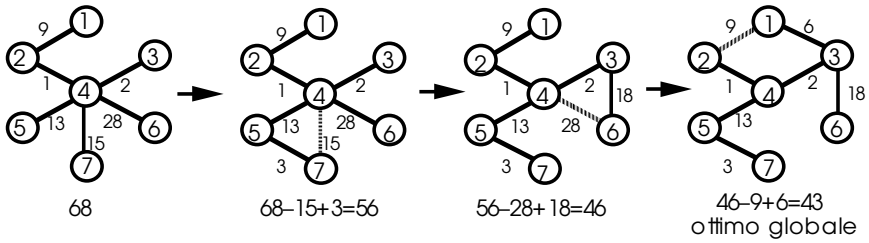


Figura 1.3: Sequenza di miglioramenti ottenuta scegliendo ad ogni passo della ricerca locale la soluzione dell'intorno che dà il massimo miglioramento.

Pertanto una procedura di ricerca locale è in grado di trovare la soluzione ottima del problema. La complessità dell'algoritmo dipende dal numero di miglioramenti. Una sequenza di miglioramenti, fino all'ottimo globale, per l'esempio di Figura 1.1, è mostrata nella Figura 1.3. Purtroppo, già la sola verifica che nessuna trasformazione possa essere effettuata, cioè che una soluzione non sia migliorabile, richiede $O(nm)$ tempo, poiché devono essere provati al più $m-n+1$ archi, e ciascuno può formare un circuito lungo al più $n$. Pertanto, l'algoritmo di Kruskal è nettamente più veloce di questo algoritmo. ■

Il più famoso algoritmo di ricerca locale è senza dubbio il metodo del *simplesso*, ideato da Dantzig nel 1947 per risolvere il problema della PROGRAMMAZIONE LINEARE [cfr. Capitolo 18]. Tale algoritmo non può però essere qui riportato perché una sua descrizione dettagliata richiede buone conoscenze matematiche di ricerca operativa. In questo capitolo, consideriamo invece la progettazione di altri due algoritmi di ricerca locale. Il primo algoritmo illustra un utilizzo piuttosto sorprendente della ricerca locale, per risolvere addirittura un problema che apparentemente non sembra neanche un problema di ottimizzazione: il solito problema dell'ordinamento degli elementi di un vettore. Il secondo invece ne mostra un utilizzo "classico", per risolvere un ben noto problema di ottimizzazione su grafi che è un caso particolare della programmazione lineare: il problema del flusso massimo.

## 1.1 Shell Sort

In una sequenza di $n$ elementi $a_1, \ldots, a_n$, un'*inversione* è data da una coppia di elementi $a_i$ e $a_j$ tali che $i < j$ e $a_i > a_j$ [cfr. Esercizio 12.3]. Il problema dell'ordinamento degli $n$ elementi può essere

formulato come problema di ottimizzazione nel modo seguente:

ORDINAMENTO. *Data la sequenza $a_1, \ldots, a_n$, trovare una permutazione degli n elementi che minimizzi il numero totale di inversioni.*

**Esempio (Inversioni)** La sequenza 1, 7, 3, 4 ha due inversioni, perché $a_2 = 7 > a_3 = 3$ e $a_2 = 7 > a_4 = 4$. La soluzione ottima 1, 3, 4, 7 ha il numero minimo di inversioni (zero), mentre la sequenza ordinata alla rovescia 7, 4, 3, 1 ne ha il numero massimo (sei). In generale, il numero minimo di inversioni per una sequenza di $n$ elementi è sempre zero, mentre quello massimo è $\sum_{i=1}^{n-1}(n-i) = (n-1)n/2$. ■

Si supponga per semplicità che gli $n$ elementi $a_1, \ldots, a_n$ siano tutti distinti, e si indichino con $\pi$ e $a_\pi$, rispettivamente, una permutazione degli indici $1, \ldots, n$ e la corrispondente permutazione degli elementi. Indichiamo inoltre con $a_\pi - \{a_i\}$ la sequenza $a_\pi$ dalla quale è stato tolto un generico elemento $a_i$. Un intorno $I(a_\pi)$ può essere definito come l'insieme di tutte le permutazioni di $a_\pi$ che, a meno di un elemento $a_i$, hanno elementi uguali nella stessa posizione relativa:

$$I(a_\pi) = \{a'_\pi : a_\pi - \{a_i\} = a'_\pi - \{a_i\}, 1 \leq i \leq n\}.$$

**Esempio (Intorno)** Si consideri la sequenza 1, 7, 3, 4; "scalando" l'1 a destra, si ottiene la sequenza 7, 1, 3, 4, indi 7, 3, 1, 4, e infine 7, 3, 4, 1. Queste quattro sequenze sono tra loro identiche, a meno dell'elemento 1: cancellando 1 in tutte, si ottiene infatti la medesima sequenza 7, 3, 4. Ragionando in modo analogo per i rimanenti elementi, si ricava: $I(1,7,3,4) = \{(7,1,3,4), (7,3,1,4), (7,3,4,1), (1,3,7,4), (1,3,4,7), (1,7,4,3), (3,1,7,4), (4,1,7,3), (1,4,7,3)\}$. ■

Un criterio di miglioramento è chiaramente quello di diminuire il numero di inversioni, cioè passare da $a_\pi$ ad $a'_\pi \in I(a_\pi)$ in modo che il numero di inversioni di $a'_\pi$ sia minore del numero di inversioni di $a_\pi$.

**Esempio (Miglioramento)** Le sequenze di $I(1,7,3,4)$ che diminuiscono il numero di inversioni sono soltanto 1, 3, 7, 4 e 1, 3, 4, 7. ■

Questo criterio di miglioramento può ricordare il semplice algoritmo di ordinamento insertionSort() [cfr. Capitolo 2].

La "pecca" dell'Insertion Sort è quella di poter spostare solo elementi tra loro adiacenti, richiedendo pertanto $O(n)$ tempo per trasferire un elemento piccolo dall'estremità destra a quella sinistra della sequenza. Il suo "pregio", d'altronde, è quello di richiedere poco tempo di elaborazione se la sequenza è "quasi" ordinata. Per eliminare la "pecca" e mantenere il "pregio", è possibile spostare elementi che non siano adiacenti, ma "distanti" $h$, con $h > 1$. È questa l'idea venuta a Shell (1959), che ha proposto un'estensione dell'Insertion Sort che porta il suo nome: Shell Sort. L'idea fondamentale consiste nell'usare una sequenza di distanze decrescenti, in modo da ordinare dapprima elementi che sono tra loro molto lontani e solo successivamente considerare elementi tra loro adiacenti che, per l'effetto degli ordinamenti precedenti, dovrebbero essere ormai "quasi" ordinati.

La seguente procedura shellSort() è simile a insertionSort(); invece di considerare elementi a distanza 1, si considerano elementi a distanza $h$, con valori di $h$ pari a ... 1093, 364, 121, 40, 13, 4, 1 (la scelta di questi particolari valori di $h$ sarà chiarita in seguito):

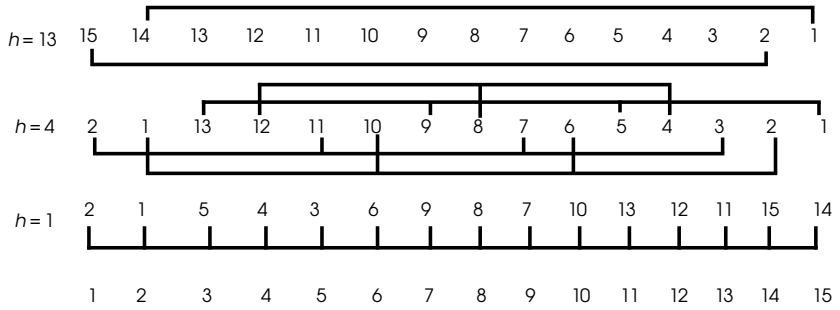


Figura 1.4: Esecuzione della procedura shellSort(), con evidenziate le sottosequenze che sono ordinate per $h = 13, 4, 1$.

```
shellSort(item[] A, int n)
    int h = 1
    while h ≤ n do h = 3·h + 1

    h = ⌊h/3⌋
    while h ≥ 1 do
        for i = h + 1 to n do
            item temp = A[i]
            int j = i
            while j > h and A[j − h] > temp do
                A[j] = A[j − h]
                j = j − h
            A[j] = temp
        h = ⌊h/3⌋
```

In pratica, la procedura riordina la sequenza di elementi con la proprietà che, scelto un elemento qualsiasi, la sottosequenza contenente un elemento ogni $h$ è riordinata. La sequenza complessiva può essere vista come formata da $h$ sottosequenze indipendenti ma "incastrate" tra loro, ognuna contenente all'incirca $n/h$ elementi. Ciascuna sottosequenza è ordinata con una ricerca locale basata sull'Insertion Sort. Per $h$ grande, sono spostati elementi della sequenza in ingresso molto "lontani" tra loro, mentre quando $h$ diventa 1 la procedura è proprio una insertionSort() sull'intera sequenza in ingresso, in cui però ci si aspetta di spostare solo pochi elementi "vicini" tra loro.

**Esempio (Shell Sort)** La Figura 1.4 mostra l'ordinamento della sequenza di 15 elementi $15,14,13,12,11,10,9,8,7,6,5,4,3,2,1$ con la procedura shellSort(). I valori di $h$ usati sono: 13, 4 ed 1. ■

La procedura shellSort() è chiaramente corretta per un qualsiasi insieme di valori per $h$ che comprenda il valore 1, perché in quest'ultimo caso essa si riduce ad insertionSort(). La sua complessità non solo dipende dalla sequenza in ingresso, ma anche, e soprattutto, dai particolari valori di $h$ usati. Un'analisi completa della complessità non può essere qui riportata, perché troppo complicata. Ci limitiamo pertanto a citare i seguenti risultati:

(1) per valori di $h$ uguali a $(3^i - 1)/2$, con $i = t, t-1, \ldots, 1$, dove $t$ è il massimo indice $s$ tale che $(3^{s+1} - 1)/2$ è maggiore di $n$ (cioè proprio per i valori ... 1093, 364, 121, 40, 13, 4, 1 utilizzati precedentemente), è stato dimostrato che shellSort() ha complessità $O(n^{1.5})$ nel

caso pessimo;
(2) per valori di $h$ uguali soltanto ad $1.72n^{1/3}$ ed 1, è stato dimostrato che shellSort() ha complessità $O(n^{5/3})$ nel caso medio;
(3) per valori di $h$ uguali a tutti gli interi della forma $3^i 2^j$ minori o uguali ad $n$, la complessità si riduce ad $O(n\log^2 n)$ nel caso pessimo;
(4) esistono altri insiemi di valori di $h$ per i quali la complessità è $O(n^{1.25})$ nel caso pessimo.

In pratica, i valori di $h$ maggiormente utilizzati sono proprio quelli del caso (1), cioè ... 1093, 364, 121, 40, 13, 4, 1, perché generabili molto facilmente. Per questi valori, c'è chi ipotizza addirittura una complessità di $O(n\log^2 n)$ oppure $O(n^{1.25})$ nel caso medio, ma nessuno finora è stato capace di provare tale congettura. Tali considerazioni mostrano come shellSort() sia in pratica una procedura di ordinamento molto efficiente, da preferire addirittura ad altri algoritmi più sofisticati, specie per $n$ non troppo elevato (per esempio $n \leq 5\,000$). I maggiori pregi di questa procedura, oltre alla facilità di programmazione e alla velocità di esecuzione, consistono nel fatto che l'ordinamento avviene iterativamente, cioè senza ricorsione, ed *in loco*, cioè senza bisogno di memoria ausiliaria né per copiare porzioni di $A$, al contrario di MergeSort(), né per gestire la ricorsione, come avviene invece sia con MergeSort() che con QuickSort().

## 1.2 Flusso massimo

Una *rete di flusso* $G = (V,E,s,p,c)$ è data da un grafo orientato $G = (V,E)$, da una coppia di vertici di $V$ detti *sorgente* $s$ e *pozzo* $p$, e da una funzione di *capacità* a valori interi positivi $c : V \times V \to \mathbf{Z}^+ \cup \{0\}$, tale per cui $c(u,v) = 0$ se $(u,v) \notin E$.

Un *flusso* in $G$ è una funzione a valori interi $f : V \times V \to \mathbf{Z}$ che soddisfa le seguenti proprietà:
(1) *Simmetria opposta*: $f(u,v) = -f(v,u)$ per ogni coppia $u,v \in V$;
(2) *Vincolo di capacità*: $f(u,v) \leq c(u,v)$ per ogni coppia $u,v \in V$;
(3) *Conservazione del flusso*: $\sum_v f(u,v) = 0$ per ogni nodo $u \in V - \{s,p\}$.

La quantità $f(u,v)$, che può essere positiva o negativa, è detta *flusso netto* dal nodo $u$ al nodo $v$, mentre il valore $|f|$ del flusso $f$ è il flusso netto totale che esce dalla sorgente (o che entra nel pozzo), ovvero $|f| = \sum_v f(s,v) = \sum_u f(u,p)$. Il problema del FLUSSO MASSIMO è definito nel modo seguente:

FLUSSO MASSIMO (MAXIMUM FLOW). *Data una rete di flusso $G = (V,E,s,p,c)$, trovare un flusso $f^*$ che ha valore $|f^*|$ più grande possibile.*

Si osservi che la proprietà di simmetria opposta implica che $f(u,u) = 0$ per ogni $u \in V$. La stessa proprietà e i vincoli di capacità implicano che se sia $(u,v) \in E$ e $(v,u) \in E$, allora $f(u,v) = f(v,u) = 0$. Pertanto, un flusso netto $f(u,v) \neq 0$ implica che $(u,v) \in E$ oppure $(v,u) \in E$. Infine, la conservazione del flusso è la stessa legge della corrente elettrica di Kirchhoff che afferma che il flusso netto totale che entra in un nodo (esclusi la sorgente ed il pozzo) eguaglia il flusso netto totale che esce dallo stesso nodo.

Il problema di trovare un flusso massimo su un grafo è intimamente correlato a quello di trovare un taglio minimo sullo stesso grafo. Dato $G$, un taglio $(S,P)$ è una partizione dell'insieme dei nodi tale che $s \in S$, $p \in P$, $S \cup P = V$, ed $S \cap P = \emptyset$. La *capacità* del taglio è $c(S,P) = \sum_{u \in S, v \in P} c(u,v)$. Un taglio di capacità minima è detto *taglio minimo*. Se $f$ è un flusso ed $(S,P)$ è un taglio, allora il flusso che attraversa il taglio è

$$f(S,P) = \sum_{u \in S, v \in P} f(u,v).$$

Una proprietà importante è che il valore di un flusso è uguale al flusso che attraversa un qualsiasi taglio, il quale a sua volta non supera la capacità del taglio stesso, cioè che $|f| = f(S,P) \leq c(S,P)$. Infatti, si ha che:

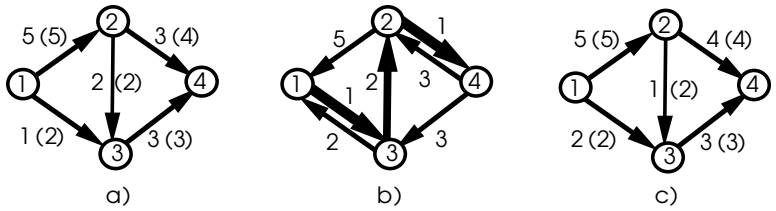


Figura 1.5: a) Un grafo $G$ con $s = 1$, $p = 4$, ed un flusso $f$ di valore $|f| = 6$, dove per ciascun arco $(u,v)$ è indicata la coppia di valori: $f(u,v)$ $(c(u,v))$. b) La rete di flusso residua $R$ ed un cammino aumentante con capacità residua $\delta = 1$. c) Il grafo $G$ con il flusso $f$ aggiornato aggiungendovi $\delta$ lungo gli archi del cammino aumentante. **TODO**: l'arco $(3,1)$ in (b) deve avere etichetta 1

- $\sum_{u\in S, v\in V} f(u,v) = \sum_{v\in V} f(s,v) + \sum_{u\in S-\{s\}, v\in V} f(u,v) = |f|$, per la proprietà di conservazione del flusso,
- $\sum_{u\in S, v\in S} f(u,v) = 0$, per la proprietà di simmetria opposta,

da cui si ricava:

$$f(S,P) = \sum_{u\in S, v\in P} f(u,v) = \sum_{u\in S, v\in V} f(u,v) - \sum_{u\in S, v\in S} f(u,v) = |f| - 0 = |f|,$$

mentre $f(S,P) \leq c(S,P)$ deriva banalmente dai vincoli di capacità.

Un importante teorema, detto del *flusso massimo / taglio minimo*, afferma che, se vale l'uguaglianza, cioè $f(S,P) = c(S,P)$, allora $f$ è un flusso massimo ed $(S,P)$ è un taglio minimo. Inoltre tale teorema dà lo spunto per progettare algoritmi di ricerca locale per trovare il flusso massimo.

Per dimostrare il teorema, si definisce *capacità residua* per un flusso $f$ la quantità $r(u,v) = c(u,v) - f(u,v)$. La rete di flusso residua $R = (V, E_r, s, t, r)$ ha lo stesso insieme $V$ di nodi di $G$, e l'insieme di archi $E_r$ contiene un arco $(u,v)$ con capacità $r(u,v)$ per ogni coppia di nodi $u,v \in V$ tale che $r(u,v) > 0$. Un *cammino aumentante* per $f$ è un cammino da $s$ a $p$ in $R$. La capacità residua $\delta$ di un cammino aumentante è uguale alla capacità residua $r(u,v)$ più piccola tra quelle di tutti gli archi $(u,v)$ che compaiono nel cammino.

È possibile definire un intorno $I(f)$ di un flusso $f$ come l'insieme di tutti i flussi ottenibili da $f$ variandone il valore, su ogni arco di un cammino aumentante, di una quantità non più grande della capacità residua del cammino aumentante stesso. In altri termini, dato un cammino aumentante per $f$, sia $g$ il flusso di valore $\delta$ tale che $g(u,v) = \delta$ e $g(v,u) = -\delta$ se l'arco $(u,v)$ appartiene al cammino aumentante, e $g(u,v) = 0$ altrimenti. Si può passare dalla soluzione $f$ ad una soluzione migliore che appartiene all'intorno $I(f)$ sommando, componente per componente, il flusso $g$ al flusso $f$, ottenendo così il flusso $f' = f + g$ di valore $|f'| = |f| + \delta$.

**Esempio (Rete di flusso residua e cammini aumentanti)** La Figura 1.5(a) mostra un grafo $G$ di quattro nodi, ove $s = 1$ e $p = 4$, ed un flusso $f$ per $G$ di valore $|f| = 6$ (per chiarezza, sono indicate soltanto le componenti del flusso relative agli archi di $G$, seguite dalle capacità tra parentesi tonde). La Figura 1.5(b) mostra invece la corrispondente rete residua $R$; ad esempio, l'arco $(2,4)$ di $R$ ha capacità residua $r(2,4) = c(2,4) - f(2,4) = 4 - 3 = 1$, mentre l'arco $(4,2)$ ha capacità residua $r(4,2) = c(4,2) - f(4,2) = 0 - (-3) = 3$. Si noti che sono comparsi alcuni archi che non esistevano in $G$, come ad esempio $(4,2)$. La Figura 1.5(b) evidenzia in grassetto anche il cammino aumentante 1, 3, 2, 4 con capacità residua $\delta = r(1,3) = r(2,4) = 1$, sul quale si può definire un flusso $g$ tale che $g(1,3) = g(3,2) = g(2,4) = \delta = 1$, $g(3,1) = g(2,3) =$

$g(4,2) = -\delta = -1$, e $g(u,v) = 0$ altrove. Sommando $g$ ad $f$ si ottiene il flusso di Figura 1.5(c), di valore $|f| = 7$. ■

È facile verificare che se $f$ è un flusso qualsiasi per $G$ ed $f^*$ è un flusso massimo per $G$, allora $f^* - f$ è un flusso massimo per $R$ di valore $|f^* - f|$. Si può quindi dimostrare il seguente teorema fondamentale:

**Teorema 1.1 (FLUSSO MASSIMO / TAGLIO MINIMO)** Le condizioni seguenti sono tra loro equivalenti:

(1) $f$ è un flusso massimo;
(2) non esiste alcun cammino aumentante per $f$;
(3) esiste un taglio $(S,P)$ tale che $|f| = c(S,P)$.

*Dimostrazione.* È immediato osservare che (1) implica (2), poiché se esistesse un cammino aumentante allora il flusso potrebbe essere aumentato e quindi non sarebbe massimo. Per verificare che (2) implica (3), siano $S$ l'insieme dei nodi di $R$ raggiungibili da $s$ con un cammino e $P = V - S$. $(S,P)$ è un taglio e poiché non esiste alcun cammino aumentante, per ogni $u \in S$ e $v \in P$ risulta che $(u,v)$ non è un arco di $R$ e quindi $c(u,v) = f(u,v)$. Ma ciò implica che $|f(S,P)| = c(S,P)$. Infine (3) implica (1), poiché da $|f| \leq c(S,P)$ per ogni flusso $f$ ed ogni taglio $(S,P)$ segue che se $|f| = c(S,P)$ allora $f$ è un flusso massimo. ■

**Esempio (continua)** Il flusso mostrato nella Figura 1.5(c) è un flusso massimo. Infatti, considerando il taglio$(S,P)$ dove $S = \{1\}$ e $P = \{2,3,4\}$, si vede immediatamente che $|f| = 7 = f(S,P) = c(S,P)$. ■

Basandosi sulla caratterizzazione matematica della soluzione fornita dal precedente teorema, sono stati proposti svariati algoritmi di ricerca locale che, partendo da un flusso $f$ nullo (cioè con $|f| = 0$), lo aumentano ripetutamente lungo cammini aumentanti finché, non esistendo più alcun cammino aumentante nel grafo residuo, viene individuato il flusso massimo. La struttura generale è mostrata nella schema di procedura maxFlow().

```
int[][] maxFlow(GRAPH G, NODE s, NODE p, int[][] c)
    NODE u,v                                          % Indici nodi
    int[][] f = new int[][]                           % Flusso parziale
    int[][] g = new int[][]                           % Flusso da cammino aumentante
    foreach u,v ∈ G.V() do
    f[u,v] = 0                                        % Inizializza un flusso nullo
    boolean stop = false
    while not stop do
        R = Rete di flusso residua del flusso f in G
        g = flusso associato ad uno o più cammini aumentanti in R
        foreach u,v ∈ G.V() do
        f[u,v] = f[u,v] + g[u,v]    % f = f + g if ∀u,v ∈ G.V() : g[u,v] = 0 then stop = true
    return f
```

L'aumento può essere fatto in base a molti criteri diversi, per esempio lungo un singolo cammino aumentante o su più cammini aumentanti contemporaneamente, preferendo quelli con massima capacità residua oppure col minimo numero di archi. Vediamo tre possibilità.

### 1.2.1 Algoritmo di Ford-Fulkerson

L'algoritmo più semplice, proposto da Ford e Fulkerson (1956), richiede semplicemente di trovare un cammino aumentante qualsiasi, utilizzando un qualunque algoritmo di visita. Assumendo che le capacità siano rappresentate da valori interi, ogni cammino aumentante incrementa il flusso almeno di 1. Poiché una visita ha costo $O(n+m)$ e possono essere effettuati al massimo $|f^*|$ incrementi, il costo di questo algoritmo è $O((n+m)|f^*|)$. Se è possibile stimare superiormente il valore del flusso massimo e questa limitazione superiore è sufficientemente bassa, un algoritmo semplice come questo può essere considerato efficiente.

### 1.2.2 Algoritmo di Edmonds-Karp

Se la ricerca del cammino aumentante è effettuata tramite una visita in ampiezza, si ottiene un algoritmo proposto da Edmonds e Karp (1972). È possibile dimostrare che la complessità dell'algoritmo è $O(nm^2)$; poiché questo algoritmo è un caso particolare di Ford e Fulkerson, è possibile calcolare entrambe le limitazioni superiori e scegliere l'algoritmo con la limitazione più favorevole.

### 1.2.3 Altri algoritmi

Gli algoritmi visti finora non sono i più efficienti in assoluto, ma sono i più semplici da spiegare. Nel sito del libro, è possibile trovare un ulteriore algoritmo chiamato dei tre indiani dalla nazionalità dei sui scopritori (Kumar, Malhotra e Maheswari), che risolve il problema con complessità $O(n^3)$. Il miglior algoritmo noto è invece stato proposto da Karmarkar nel 1984 e ha un complessità di $O(mn\log(n^2/m))$.

## 1.3 Reality check

Un enorme numero di problemi può essere formulato come un problema di flusso massimo; ne potete vedere un paio di esempi negli esercizi (Esercizio 1.9 per gli abbinamenti bipartiti e Esercizio 1.10 per una misura dell'arco connettività). Alcuni autori (Skiena, 2008) arrivano a suggerire di progettare grafi, non algoritmi: utilizzando soluzioni esistenti di problemi ben noti.[1] Le reti di flusso possono essere utilizzate per modellare reti di trasporto, reti di telecomunicazioni, servizi logistici e molti altri ancora. La loro applicabilità è così vasta che risulta difficile identificare un unico esempio.

## 1.4 Esercizi

**Esercizio 1.1** (Tagli minimi)**.** Si elenchino tutti i tagli di capacità minima del grafo illustrato in Figura 1.5(a) per la sorgente $s = 1$ e il pozzo $p = 4$.

**Esercizio 1.2** (Scheduling)**.** Dati $n$ programmi con tempi di esecuzione $t_1, \ldots, t_n$, si vuole eseguirli su un processore in modo da minimizzarne il tempo medio di completamento $(1/n)\sum_i c_i$, dove il tempo di completamento $c_i$ del programma $i$ è la somma del suo tempo di esecuzione e dei tempi di esecuzione di tutti i programmi che lo precedono nell'ordinamento (schedule) finale. Si progetti un algoritmo di ricerca locale per risolvere il problema.

**Esercizio 1.3** (Ciclo euleriano)**.** Dato un grafo orientato $G = (V,E)$, si vuole trovare un *ciclo euleriano*, cioè un cammino chiuso (ma non semplice) che include ciascun arco una e una sola volta. Si determini una condizione necessaria e sufficiente affinché $G$ ammetta un ciclo euleriano, e si progetti un algoritmo di ricerca locale per trovarlo.

[1] Si suol dire scherzosamente che i matematici non risolvono i problemi, ma li trasformano in altri problemi dei quali conoscono già le soluzioni.

**Esercizio 1.4** (Riempimento di matrice)**.** Dati $2n$ interi non negativi $c_1, c_2, \ldots, c_n, r_1, r_2, \ldots, r_n$, si vuole determinare una matrice $n \times n$ con elementi interi non negativi tali che la somma degli elementi della colonna $i$-esima sia uguale a $c_i$ e la somma degli elementi della riga $i$-esima sia uguale ad $r_i$. Quali condizioni devono valere su $c_1, \ldots, c_n, r_1, \ldots, r_n$ affinché esista una soluzione? Si formuli il problema come un problema di flusso massimo. Si fornisca poi un algoritmo di complessità $O(n^2)$ senza usare il flusso massimo.

**Esercizio 1.5** (Somma di matrici)**.** Siano dati una matrice $M$ di dimensione $m \times n$ con elementi non negativi e quattro interi non negativi $l_1$, $l_2$, $t_1$, $t_2$. Si vogliono determinare due matrici $M_1$ e $M_2$ di dimensione $m \times n$ con elementi interi non negativi tali che $M = M_1 + M_2$, la somma degli elementi di ogni linea (cioè riga o colonna) di $M_i$ non superi $l_i$, e la somma di tutti gli elementi di $M_i$ non superi $t_i$, per $i = 1,2$. Si formuli il problema come un problema di flusso massimo.

**Esercizio 1.6** (Formulazioni flusso massimo)**.** Si dimostri che le seguenti varianti del problema del flusso massimo possono essere ricondotte alla formulazione originale del problema descritta nel presente capitolo:

(1) Il grafo ha più di una sorgente e più di un pozzo;
(2) I nodi, oltre agli archi, hanno capacità;
(3) Il grafo non è orientato.

**Esercizio 1.7** (Cammini indipendenti)**.** Dato un grafo orientato $G = (V,E)$ e due vertici $u, v$ contenuti in $V$, trovare il numero totale di cammini "edge-independent", ovvero in cui un arco può comparire al massimo in un cammino.

**Esercizio 1.8** (Intorno per abbinamento massimo)**.** Dato un grafo non orientato, un *abbinamento* (*matching*, in inglese) $M$ è un sottoinsieme di archi tale che non esistono due archi incidenti nel medesimo nodo. Si definisca un intorno per il problema di trovare un abbinamento di massima cardinalità.

**Esercizio 1.9** (Abbinamento bipartito)**.** Dato un grafo non orientato bipartito, il problema dell'ABBINAMENTO BIPARTITO consiste nel trovare un abbinamento di massima cardinalità [cfr. Esercizio 1.8]. Si dimostri che il problema è risolubile con l'algoritmo dei tre indiani con complessità $O(m\sqrt{n})$.

**Esercizio 1.10** (Connettività)**.** La *arco-connettività* di un grafo non orientato e connesso è la dimensione del più piccolo insieme di archi la cui rimozione causa la divisione del grafo in due componenti connesse separate. Si progetti un algoritmo per calcolare l'arco-connettività di un grafo.

## 1.5 Soluzioni

**Soluzione Esercizio 1.1**

I seguenti tagli hanno capacità minima:

- $S = \{1\}, P = \{2,3,4,5,6\}$
- $S = \{1,4\}, P = \{2,3,5,6\}$
- $S = \{1,2,3,4,5\}$, $P = \{6\}$
- $S = \{1,2,4,5\}$, $P = \{3,6\}$

**Soluzione Esercizio 1.3**

Il famoso teorema di Eulero assicura che $G$ contiene un ciclo euleriano se e solo se è connesso (trascurando l'orientamento degli archi), e per ciascun nodo il numero di archi entranti nel nodo è uguale al numero di archi uscenti dal nodo. Si assuma quindi che $G$ verifichi tali condizioni.

Una *partizione di Eulero* $P$ per $G$ è una partizione di $E$ in cammini chiusi (non necessariamente semplici) disgiunti sugli archi. Una partizione di Eulero può essere determinata specificando per

ogni arco $a = (u,v)$ un unico successore $\mathsf{succ}(a) = b = (v,w)$ in modo che non esistano mai due archi con lo stesso successore. Ovviamente, la cardinalità di una partizione di Eulero è compresa tra 1 ed $\lfloor n/2 \rfloor$ ed un ciclo euleriano corrisponde ad una partizione di Eulero di minima cardinalità. Una partizione iniziale $P$ è facilissima da trovare, per esempio considerando un nodo alla volta di $G$ e definendo l'$h$-esimo arco uscente dal nodo come il successore dell'$h$-esimo arco entrante. Dati due archi $a$ e $b$, con $succ(a) = c$ e $\mathsf{succ}(b) = d$, si può definire un'operazione $\mathsf{scambio}(a,b)$ che scambia tra loro i successori in modo che $\mathsf{succ}(a) = d$ e $\mathsf{succ}(b) = c$. Si può allora definire l'intorno

$$I(P) = \{P' : P' \text{ è ottenuta da } P \text{ con una operazione } \mathsf{scambio}\}.$$

Ovviamente, uno scambio può far aumentare o diminuire di uno la cardinalità di una partizione, a seconda che divida un cammino chiuso in due cammini disgiunti sugli archi, oppure che fonda due cammini chiusi in uno solo. L'algoritmo di ricerca locale sceglierà ad ogni iterazione un'operazione di scambio che decrementi la cardinalità della partizione e si fermerà quando non saranno più possibili miglioramenti, cioè quando sarà stato individuato un ciclo euleriano.

**Soluzione Esercizio 1.7**
Si costruisce una funzione di capacità $c$ tale per cui $c(x,y) = 1$ se $(x,y) \in E$, mentre $c(x,y) = 0$ se $(x,y) \notin E$. Si consideri il massimo flusso fra $u$ e $v$; questo valore corrisponde al numero totale di cammini indipendenti, in quanto essendo la capacità di tutti gli archi pari ad 1, ogni cammino aumentante avrà valore di flusso 1 e tutti i suoi archi saranno distinti dagli altri cammini aumentanti.

**Soluzione Esercizio 1.8**
Sia dato un abbinamento qualsiasi $M$. Si definisca un nodo "scoperto" se non ha alcun arco di $M$ incidente in esso. Si definisca "catena alternante" una catena di archi costituita, alternatamente, da un arco non appartenente ad $M$ e da uno appartenente ad $M$. Si definisca "catena aumentante" una catena alternante tra due nodi scoperti. In una catena aumentante, il numero di archi che non fanno parte dell'accoppiamento è uno più degli archi che ne fanno parte. Pertanto, si può definire un intorno

$$\begin{aligned} I(M) = \{&M' : M' \text{ è ottenibile da } M \text{ scambiando in una catena aumentante gli archi.} \\ &\text{che fanno parte dell'accoppiamento con gli altri}\} \end{aligned}$$

(Trovare una catena aumentante richiede tempo polinomiale, ma l'algoritmo non è affatto banale).

**Soluzione Esercizio 1.9**
Sia $B = (V,E)$ un grafo non orientato bipartito. Per definizione, esiste una partizione dell'insieme $V$ dei nodi in due insiemi disgiunti $S$ e $P$ tale che per ogni arco $[u,v]$ risulta $u \in S$ e $v \in P$ [cfr. Esercizi 9.3 e 9.4]. Si consideri un grafo orientato $G$ con capacità unitarie sugli archi, ottenuto da $B$ orientando gli archi da $S$ verso $P$, aggiungendo un nodo sorgente $s$ collegato con un arco ad ogni nodo di $S$ ed un nodo pozzo $p$ collegato con un arco da ogni nodo di $P$. Formalmente, si costruisce $G = (V',E')$ tale che:

$$\begin{aligned} V' &= V \cup \{s,p\} \\ E' &= \{(u,v) : [u,v] \in E, u \in S, v \in P\} \cup \{(s,u) : u \in S\} \cup \{(v,p) : v \in P\} \\ c(u,v) &= 1 \text{ per ogni } (u,v) \in E'. \end{aligned}$$

Come illustrato nella Figura 1.6, è facile verificare che un flusso massimo $f^*$ di valore $|f^*|$ in $G$ corrisponde ad un abbinamento $M$ di cardinalità $|f^*|$ in $B$, dove gli archi $[u,v] \in M$ sono dati dagli archi $(u,v)$ con $u \in S$, $v \in P$ ed $f^*(u,v) = 1$.

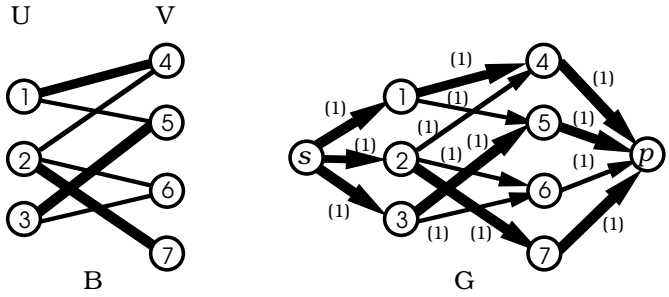


Figura 1.6: Un grafo non orientato bipartito $B$ ed il corrispondente grafo orientato $G$ con capacità unitaria (tra parentesi). Gli archi in grassetto evidenziano un abbinamento massimo (di cardinalità 3) di $B$ e le corrispondenti componenti unitarie del flusso massimo di $G$.

Per dimostrare che l'algoritmo dei 3 indiani ha complessità $O(m\sqrt{n})$ sul grafo $G$ così definito, mostriamo che la procedura satura() richiede $O(m)$ tempo e che kmn() effettua $O(\sqrt{n})$ iterazioni in ciascuna delle quali cerca un flusso saturante.

Poiché ogni arco del grafo $G$ ha capacità unitaria, anche il grafo residuo $R$ ha capacità unitarie. Inoltre, dato che il flusso ha componenti intere, non ci sono mai riempimenti senza saturazioni, ed ogni arco di $R$ o è saturato e cancellato oppure non è neanche esaminato. Pertanto, il tempo richiesto da satura() è $O(m)$. Vediamo che il numero di volte che è cercato un flusso saturante è al più $2\lceil\sqrt{n-2}\rceil$. Siano $f$ il flusso ad una generica iterazione, $f^*$ il flusso massimo, ed $R$ il grafo residuo per $f$. Poiché $f^* - f$ è un flusso per $R$ ed ogni componente del flusso vale 0 o 1 su ogni arco, è possibile partizionare gli archi di $R$ per i quali il flusso è 1 in un insieme di $|f^* - f|$ cammini tra $s$ e $p$ (più qualche eventuale ciclo). Poiché le capacità residue sono unitarie ed ogni nodo di $R$ ha al più un arco entrante oppure al più un arco uscente, ogni nodo (tranne $s$ e $p$) appare in al più uno di tali cammini, e pertanto $f$ ha un cammino aumentante di lunghezza $k \leq (n-2)/|f^* - f| + 1$. Ma dopo $2\lceil\sqrt{n-2}\rceil$ iterazioni, $k \geq \sqrt{n-2}+1$, perché ad ogni iterazione la lunghezza $k$ dei cammini aumentanti più corti aumenta. Combinando quest'ultime due relazioni, si ricava che $|f^* - f| \leq \sqrt{n-2}$ e quindi dopo al più altre $\lfloor\sqrt{n-2}\rfloor$ iterazioni viene trovato il flusso massimo. Pertanto, il numero di iterazioni effettuate da kmn() è $O(\sqrt{n})$.

**Soluzione Esercizio 1.10**
Si costruisca una versione orientata $G' = (V, E')$ del grafo, dove ad ogni arco $[u, v]$ in $E$ corrispondono due archi $(u, v)$ e $(v, u)$ in $E'$. $G'$ ha $n$ vertici e $2m$ archi. Si associ una funzione di capacità $c$ tale che per ogni arco $(u, v) \in E'$, $c(u, v) = 1$.

Sia $f^*(u, v)$ il flusso massimo fra $u$ e $v$ in $G'$, per ogni $u$ e $v$. Per il teorema Flusso massimo / Taglio minimo, $f^*(u, v)$ è la dimensione del minimo insieme di nodi che separa $u$ da $v$. Scegliamo un nodo $u$ ed eseguiamo l'algoritmo per calcolare $f^*(u, v)$ per tutti i nodi $v \neq u$. L'arco connettività è il minimo valore $|c^*| = \min_{v \neq u} |f^*(u, v)|$. La complessità è $O(n^2(n+m))$, in quanto si ripete $n$ volte l'algoritmo di Ford-Fulkerson di costo $|f^*|(n+m)$, con $|f^*| = O(n)$.