Indice

Indice degli Algoritmi

Capitolo I

Algoritmi ricorsivi e analisi della loro complessità

...

Capitolo II

Algoritmi randomizzati e loro analisi di prestazione

...

Capitolo III

Tecniche di valutazione di complessità ammortizzata

L'*analisi ammortizzata* è una tecnica di valutazione della complessità computazionale. Essa considera una sequenza di operazioni svolte nell'esecuzione di un certo algoritmo, attribuendo ad ognuno dei singoli passi elementari di esso (o comunque a delle sottoparti di esso) un costo computazionale *medio*. Sebbene si parli di complessità media dei vari passi, l'analisi ammortizzata non ha nulla a che vedere con le tecniche di analisi di tipo probabilistico, già trattate in precedenza. Le valutazioni che scaturiranno riguarderanno il caso pessimo dell'esecuzione dell'algoritmo, dunque come risultato si garantirà un certo costo medio di complessità per ogni operazione nel caso pessimo.

Esistono diversi approcci all'analisi ammortizzata, di seguito ne mostreremo tre diversi: l'*analisi aggregata*, il *metodo della contabilità*, il *metodo del potenziale*. Ognuno di essi porterà ad una diversa nozione di *costo ammortizzato*. Prima di entrare nel dettaglio dei diversi metodi presenteremo due casi di studio elementari, sui quali svolgeremo a titolo di esempio le valutazioni di complessità usando le diverse tecniche.

III.1 Casi di studio

III.1.1 Operazioni sulla pila

Una *pila* (*stack*) è una struttura dati costituita da una lista semplice S, idealmente di capacità infinita, sulla quale si possono effettuare inserimenti ed estrazioni di elementi secondo la ben nota politica *LIFO* (*Last In First Out*). Dunque numerando le posizioni della pila da 1 in avanti e indicando con $S[i]$ l'i-esima posizione nella pila, se in un certo istante essa contiene n elementi si potranno svolgere unicamente le seguenti operazioni:

- PUSH(S, x), il cui effetto è quello di inserire l'elemento x sopra la *cima* alla pila, ovvero in posizione $n + 1$; ora l'elemento x costitusce la nuova cima della pila;

- POP(S), la quale restituisce l'elemento in cima alla pila, ovvero l'elemento in po-

sizione n; tale lettura *consuma* la cima della pila, per cui la nuova cima sarà costituita dall'elemento in posizione $n-1$.

In realtà considereremo una piccola estensione di questo modello, aggiungendo l'operazione MULTIPOP, la quale riceve come parametri la pila S che si considera e un intero $k > 0$, e banalmente effettua k volte l'operazione di POP su S, terminando l'esecuzione se viene svuotata completamente la pila prima di completare le k estrazioni. Per completezza mostriamo lo pseudocodice della MULTIPOP, nel quale si utilizza l'operazione booleana PILAVUOTA con ovvio significato.

Algoritmo III.1 MULTIPOP(S, k)

```
1: while not PilaVuota(S) and k ≠ 0 do
2:     Pop(S)
3:     k ← k − 1
```

III.1.2 Incremento di un contatore binario

Un *contatore binario* A non è altro che una sequenza di k bit, la quale rappresenta la codifica binaria di un numero intero positivo. Per essere più precisi, numeriamo i bit del contatore binario da 0 a $k-1$, indicando al solito con $A[i]$ il bit nell'i-esima posizione. Per quanto riguarda la convenzione della rappresentazione, sia $A[0]$ il bit meno significativo della stringa, in modo che essa rappresenti il numero naturale

$$\sum_{i=0}^{k-1} A[i] \cdot 2^i.$$

Sulla struttura dati descritta definiamo l'operazione INCREMENTO, la quale somma (in binario) 1 alla sequenza A fornita in ingresso. Si noti che invocando INCREMENTO su un contatore A costituito da tutti 1 si verifica il cosiddetto *overflow*, a seguito del quale il contatore viene azzerato. Di seguito mostriamo lo pseudocodice, nel quale si utilizza l'operazione LENGTH che restituisce il numero di bit del contatore binario passatogli in ingresso.

Algoritmo III.2 INCREMENTO(A)

```
1: i ← 0
2: while i < Length(A) and A[i] = 1 do
3:     A[i] ← 0
4:     i ← i + 1
5: if i < Length(A) then
6:     A[i] ← 1
```

III.2 Approcci all'analisi ammortizzata

III.2.1 Analisi aggregata

Mediante l'*analisi aggregata*, se si mostra che, per ogni n intero, un'arbitraria sequenza di operazioni nel caso pessimo impiega $T(n)$ unità di tempo (dove $T(n)$ è in genere espresso in termini asintotici mediante la relazione O), allora si attribuisce ad ogni operazione un *costo ammortizzato* di $T(n)/n$. Si noti che non è richiesta omogeneità fra le operazioni, ovvero non è richiesto che le varie operazioni compiute siano dello stesso tipo. Ognuna di esse avrà in generale un costo *vero* diverso, ma la somma dei costi ammortizzati di una sequenza di operazioni dovrà essere un upper bound del costo vero della sequenza. Questa è evidentemente una proprietà irrinunciabile, e come vedremo è rispettata da tutte le tecniche di analisi ammortizzata.

Nei prossimi esempi applichiamo l'analisi aggregata ai casi di studio presentati nella sezione precedente.

Esempio III.1 (Operazioni sulla pila). Nel seguito, con $|S|$ indicheremo il numero di elementi attualmente presenti nella pila. Valutiamo dapprima la complessità delle operazioni disponibili singolarmente. Per PUSH e POP è banale osservare che richiedono tempo d'esecuzione costante, dunque hanno entrambe complessità $O(1)$. La MULTIPOP invoca k volte la POP se la pila S contiene almeno k elementi, in caso contrario il numero di invocazioni scende a $|S|$: in sintesi la complessità della MULTIPOP risulta $O(\min(|S|, k))$, che nel caso pessimo diventa $O(k)$.

Consideriamo ora un'arbitraria sequenza di n operazioni sulla pila S inizialmente vuota. Per tutta la durata della sequenza la dimensione della pila si mantiene inferiore a n, dunque l'operazione più costosa invocabile risulta essere una MULTIPOP(S, m), con $m < n$ e dunque complessità dell'ordine di $O(n)$. Con un'analisi estremamente grossolana si potrebbe affermare che nel caso pessimo la sequenza non può costare più di n invocazioni di MULTIPOP(S, n), dunque il costo complessivo è certamente limitato superiormente da $O(n^2)$.

É ovvio che si può fare una stima migliore: partendo dalla pila vuota non si può far altro che inserire n_1 elementi nella pila, ed estrarne n_2, dove $n_1 \geq n_2$ e $n_1 + n_2 = n$, da cui si deduce che il costo della sequenza corrisponde al costo di una sequenza con esattamente n_1 PUSH ed n_2 POP, in totale $O(n)$. Assegnamo allora alle tre operazioni disponibili un costo ammortizzato di $O(n/n) = O(1)$. ■

Esempio III.2 (Contatore binario). Poniamo $k =$ LENGTH(A). La singola operazione INCREMENTO, nel caso pessimo, che si ottiene invocandola su un contatore A avente le $(k-1)$ cifre meno significative a 1, deve effettuare k inversioni di bit, per un costo complessivo di $O(k)$. Con un'analisi grossolana possiamo dire che una sequenza di n INCREMENTO ha una complessità dell'ordine di $O(nk)$. Anche in questo caso è possibile essere più precisi, osservando che il bit $A[0]$ viene invertito ad ogni invocazione, il bit $A[1]$ viene invertito $\lfloor n/2 \rfloor$ volte, ed in generale il bit $A[i]$ viene invertito $\lfloor n/2^i \rfloor$ volte, per $i = 0, ..., \lfloor \lg n \rfloor$. Il numero totale di operazioni di inversione (ognuna dal costo di $O(1)$) risulta

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor .$$

Come di consueto, è possibile maggiorare la sommatoria (che è a termini positivi) rinunciando ad arrotondare all'intero inferiore, e passando ad una serie geometrica (convergente, essendo la

ragione $1/2 < 1$):

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = n \, \frac{1}{1 - 1/2} = 2n.$$

Dunque nel caso pessimo la complessità temporale di esecuzione della sequenza è $O(n)$, per cui assegnamo ad ogni singola INCREMENTO un costo ammortizzato di $O(n/n) = O(1)$. ■

III.2.2 Metodo della contabilità

Il *metodo della contabilità* consiste nell'assegnare un *costo ammortizzato* ad ogni operazione, come somma del costo attuale dell'operazione e di un *credito*, scelto in maniera opportuna. In generale operazioni diverse potranno avere un costo ammortizzato maggiore o minore del loro costo attuale. Perché un simile procedimento abbia un senso è necessario imporre dei vincoli all'attribuzione dei crediti alle varie operazioni.

Data un'operazione i appartenente ad un certo dominio considerato di operazioni, indichiamo con c_i il suo costo attuale e con $\widehat{c}_i$ il suo costo ammortizzato. Dunque la quantità $\widehat{c}_i - c_i$ è pari al credito attribuito all'operazione i, e può essere positiva o negativa. La richiesta da fare a questo punto è che per ogni $n > 0$ e per ogni sequenza ammissibile di n operazioni $i_1, i_2, ..., i_n$ risulti

$$\sum_{k=0}^{n} \widehat{c}_{i_k} \geq \sum_{k=0}^{n} c_{i_k}, \tag{III.1}$$

ovvero si richiede che per ogni sequenza ammissibile di n operazioni il bilancio complessivo dei crediti assegnati sia non negativo. Così facendo, qualora risultasse che il costo ammortizzato di una certa sequenza di operazioni sia $O(f(n))$ per una certa funzione f, allora tale upper bound si applicherà anche al costo attuale della sequenza. Si noti che il vincolo (III.1) impone che non siano ammissibili tutte le sequenze di una sola operazione che abbia credito negativo.

Di seguito applichiamo il metodo della contabilità ai due casi di studio di cui sopra.

Esempio III.3 (Operazioni sulla pila). Abbiamo già discusso la complessità attuale delle tre operazioni disponibili, che risulta

$$c_{\text{PUSH}} = c_{\text{POP}} = 1, \qquad c_{\text{MULTIPOP}} = \min(|S|, k).$$

Una buona assegnazione dei costi ammortizzati è la seguente

$$\widehat{c}_{\text{PUSH}} = 2, \qquad \widehat{c}_{\text{POP}} = \widehat{c}_{\text{MULTIPOP}} = 0.$$

Ad ogni PUSH attribuiamo un costo ammortizzato costituito dal suo costo attuale più un'unità ulteriore che viene accumulata come credito, mentre consideriamo "gratuite" (in termini di costo ammortizzato) le operazioni di estrazione dalla pila. É banale osservare che tutte le sequenze ammissibili di operazioni a partire dalla pila vuota rispettano la (III.1), infatti il numero delle POP (dove k POP(S) equivalgono ad una MULTIPOP(S, k)) nella sequenza non può superare quello delle PUSH, quindi il costo delle PUSH viene pagato con un'unità del credito accumulato fino al momento dell'invocazione. Dunque, per ogni sequenza di n PUSH, POP e MULTIPOP il costo ammortizzato è dell'ordine di $O(n)$, ed essendo un upper bound del costo attuale anche quest'ultimo sarà di quest'ordine. ■

Esempio III.4 (Contatore binario). Abbiamo già osservato che il costo della INCREMENTO è proporzionale al numero di bit invertiti. Naturalmente, in tutte le sequenze ammissibili di INCREMENTO a partire dal contatore azzerato, ogni bit può essere rimesso a 0 solo se in precedenza era stato messo ad 1. Dunque potremmo pensare di assegnare un costo ammortizzato di 2 all'operazione di assegnamento ad 1 di un bit, e considerare gratuita l'operazione di assegnamento a 0 del bit: il costo attuale di questa viene pagato utilizzando il credito residuo accumulato quando il bit era stato posto a 1. In sintesi:

$$\widehat{c}_{\leftarrow 1} = 2, \quad \widehat{c}_{\leftarrow 0} = 0,$$

dunque ogni INCREMENTO ha un costo ammortizzato minore[1] o uguale a 2. Ogni sequenza di n INCREMENTO ha un costo ammortizzato dell'ordine di $O(n)$, che risulta un upper bound anche del costo attuale. ■

III.2.3 Metodo del potenziale

Il *metodo del potenziale* dell'analisi ammortizzata costituisce una sottile modifica del precedente metodo. La differenza consiste nel fatto che, questa volta, il "lavoro prepagato" (ciò che nella precedente sezione chiamavamo credito) viene conteggiato come una sorta di "energia potenziale" associata alla struttura dati, che verrà rilasciata per pagare operazioni future.

Indichiamo con D_i lo stato della struttura dati considerata dopo l'esecuzione di i operazioni, dunque D_0 rappresentà lo stato iniziale. Con c_i indichiamo il costo attuale della i-esima operazione invocata nella sequenza di esecuzione considerata. Una funzione *potenziale* Φ mappa lo stato D_i della struttura dati in un numero reale non negativo $\Phi(D_i)$, che è appunto il potenziale associato a D_i. Il *costo ammortizzato* $\widehat{c}_i$ della i-esima operazione è definito come

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}),$$

dove il termine $\Phi(D_i) - \Phi(D_{i-1})$ rappresenta la differenza di potenziale osservata nel passare da D_{i-1} a D_i. Un conto immediato ci fornisce il costo ammortizzato di una sequenza di n operazioni

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \\ &= \left(\sum_{i=1}^{n} c_i\right) + \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1})) = \\ &= \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

Da questo risultato appare chiaro che per rispettare il vincolo (III.1) discusso nella sezione precedente è necessario definire un potenziale Φ tale che, per ogni sequenza ammissibile

[1] Quando l'invocazione di INCREMENTO causa un overflow del contatore, il costo ammortizzato sarebbe 0.

di n operazioni, risulti $\Phi(D_n) > \Phi(D_0)$. Si noti che, analogamente a quanto accade con i potenziali in fisica, non è importante il valore di $\Phi(D_i)$ di per sè, quanto piuttosto la differenza di potenziale $\Phi(D_i) - \Phi(D_{i-1})$, dunque conviene spesso porre $\Phi(D_0) = 0$, ed assegnare gli altri valori di Φ di conseguenza.

Esempio III.5 (Operazioni sulla pila). Una ragionevole funzione potenziale per il caso della pila è

$$\Phi(S_i) = \text{n}^\circ \text{ di elementi nella pila } S_i,$$

infatti risulta $\Phi(S_0) = 0$, ed è ovvio che il potenziale non potra mai scendere al di sotto di $\Phi(S_0)$. Calcoliamo il costo ammortizzato delle varie operazioni. Supponiamo che la i-esima operazione sia una PUSH, allora la differenza di potenziale risulta

$$\Phi(S_i) - \Phi(S_{i-1}) = (|S_{i-1}| + 1) - |S_{i-1}| = 1,$$

da cui

$$\widehat{c}_{\text{PUSH}} = c_{\text{PUSH}} + \Phi(S_i) - \Phi(S_{i-1}) = 1 + 1 = 2$$

Supponiamo ora che la j-esima operazione sia un'invocazione di MULTIPOP(S_{j-1}, k) (in particolare una POP se $k = 1$), la differenza di potenziale questa volta risulta

$$\Phi(S_j) - \Phi(S_{j-1}) = \min(|S_{j-1}|, k),$$

da cui

$$\widehat{c}_{\text{MULTIPOP}} = c_{\text{MULTIPOP}} + \Phi(S_j) - \Phi(S_{j-1}) = \min(|S_{j-1}|, k) - \min(|S_{j-1}|, k) = 0$$

Il costo ammortizzato delle singole operazioni è $O(1)$, da cui ogni sequenza ha un costo ammortizzato (e dunque anche attuale) dell'ordine di $O(n)$. ■

Esempio III.6 (Contatore binario). Definiamo il potenziale nel modo seguente

$$\Phi(A_i) = \text{n}^\circ \text{ di 1 nella sequenza binaria del contatore } A_i.$$

Consideriamo al solito di partire con il contatore azzerato e supponiamo che nella sequenza considerata l'i-esima INCREMENTO metta a 0 t_i bit del contatore A_{i-1}, e metta a 1 più un solo bit. Risulta allora

$$\Phi(A_i) \leq \Phi(A_{i-1}) + 1 - t_i,$$

ovvero

$$\Phi(A_i) - \Phi(A_{i-1}) \leq 1 - t_i.$$

Il costo ammortizzato della i-esima INCREMENTO è pari a

$$\widehat{c}_i = c_i + \Phi(A_i) - \Phi(A_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2,$$

che risulta indipendente dall'indice i, ovvero ogni INCREMENTO nella sequenza ha una complessità ammortizzata dell'ordine di $O(1)$. Dunque la complessità ammortizzata di una sequenza di n operazioni è dell'ordine di $O(n)$, limite che si applica anche alla complessità attuale. ■

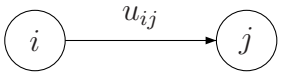


Figura III.1. Rappresentazione di un grafo.

III.3 Applicazioni al problema del flusso massimo

In questa sezione presentiamo un'applicazione dell'analisi ammortizzata per la valutazione della complessità computazionale di alcuni algoritmi per il calcolo del flusso massimo in un grafo. Cominceremo con dei richiami sulla descrizione del problema e sugli algoritmi risolutivi, per poi passare alla loro analisi ammortizzata.

III.3.1 Il problema del flusso massimo

Un *grafo orientato* (o semplicemente *grafo*) G è costituito da una coppia (N, A), dove N è l'insieme dei *nodi*, mentre $A \subseteq N \times N$ è l'insieme degli *archi*. Si dicono rispettivamente *stella entrante* e *stella uscente* del nodo i i seguenti insiemi

$$\mathrm{BS}(i) = \{(j, i) \in A\}$$
$$\mathrm{FS}(i) = \{(i, j) \in A\}$$

Ad ogni arco $(i, j) \in A$ è possibile associare una *capacità* $u_{ij} \geq 0$, che indica il flusso massimo di un certo bene che può scorrere nell'arco (i, j). Ancora, è possibile associare agli archi una quantità $x_{ij} \geq 0$ che costituisce il *flusso* assegnato all'arco (i, j). Capacità e flusso sugli archi del grafo sono rappresentati sinteticamente con i vettori

$$\boldsymbol{u} = (u_{ij})_{(i,j)\in A}, \qquad \boldsymbol{x} = (x_{ij})_{(i,j)\in A}$$

Un flusso $\boldsymbol{x}$ è detto *ammissibile* se rispetta le seguenti:

$$0 \leq x_{ij} \leq u_{ij}, \ \forall (i, j) \in A \qquad (\textit{Vincoli di capacità})$$
$$\sum_{(j,i)\in \mathrm{BS}(i)} x_{ji} = \sum_{(i,j)\in \mathrm{FS}(i)} x_{ij}, \ \forall i \in N \qquad (\textit{Vincoli di conservazione del flusso})$$

La figura III.1 mostra la notazione impiegata per rappresentare un grafo.

Descriviamo ora il problema del flusso massimo.

Problema del flusso massimo. Dato un grafo $G = (N, A)$ con i nodi $s, t \in N$ detti rispettivamente *sorgente* e *destinazione* tali che

$$\mathrm{BS}(s) = \mathrm{FS}(t) = \{(t, s)\}, \qquad u_{ts} = \infty,$$

il problema del flusso massimo consiste nel trovare un flusso ammissibile $\boldsymbol{x}$ per il grafo G che massimizzi x_{ts}. □

Informalmente parlando, l'arco (t, s) è un arco *fittizio* che serve solo a descrivere il problema in maniera consistente rispetto ai vincoli di capacità e conservazione del flusso. L'interpretazione da dare al problema è che il nodo s *generi* un flusso di bene, mentre il nodo t *consumi* il flusso. La quantità x_{ts} rappresenta allora l'ammontare complessivo del flusso spedito da s verso t.

Vediamo ora un'applicazione del problema del flusso massimo.

Esempio III.7 (Assegnazione dei lavori alle macchine). Suddividiamo la giornata lavorativa di un'officina in k slot temporali $T_1, ..., T_k$ della stessa durata, ad esempio si può pensare che ognuno dei T_i rappresenti un diverso intervallo di un'ora nell'arco della giornata. Supponiamo che l'officina possegga n macchine uguali e che abbia una lista di m lavori $L_1, ..., L_m$ commissionati per la giornata. Ogni lavoro L_i è descritto da una tripla (p_i, r_i, d_i), dove

$$\begin{array}{ll} & p_i = \text{n}^\circ \text{ slot di tempo necessari per completare il lavoro,} \\ \text{(III.2)} & r_i = \text{slot di tempo in cui può iniziare il lavoro,} \\ & d_i = \text{slot di tempo entro cui deve essere terminato il lavoro.} \end{array}$$

Supponiamo che una macchina possa interrompere un lavoro assegnatogli al termine di uno slot di tempo, per poi riprenderlo in futuro, inoltre supponiamo che non sia possibile assegnare uno stesso lavoro contemporaneamente a più macchine. Cerchiamo di ricondurre il problema dell'assegnamento dei lavori nel rispetto dei vincoli (III.2) ad un opportuno problema di flusso massimo.

Costruiamo un grafo $G = (N, A)$ nel seguente modo:

- $N = \{s, t, L_1, L_2, ..., L_m, T_1, T_2, ...T_k\}$;
- per ogni lavoro L_i esiste un arco (s, L_i) avente capacità $u_{sL_i} = p_i$, ed un arco (L_i, T_j) di capacità 1 per ogni slot di tempo T_j compreso tra gli slot r_i e d_i; inoltre per ogni slot di tempo T_j esiste un arco (T_j, t) di capacità m pari al numero di macchine dell'officina; non dimentichiamo infine l'arco fittizio (t, s) di capacità $u_{ts} = \infty$.

Si intuisce immediatamente che il bene che fluisce lungo gli archi del grafo così realizzato è un flusso di *lavoro*. Se il flusso massimo $\boldsymbol{x}$ è tale da saturare tutti gli archi uscenti da s, ovvero $x_{sL_j} = p_j$ per ogni L_j, allora è possibile soddisfare tutti i vincoli e portare a termine tutti i lavori commissionati.

Supponiamo che il carico di lavoro sia tale da non consentire di soddisfare tutte le richieste. L'officina potrebbe ricorrere a diverse soluzioni per far fronte al fabbisogno: tipicamente potrebbe affidare parte dei lavori ad un'officina esterna, oppure potrebbe incrementare la propria capacità produttiva acquistando nuovi macchinari. Può allora essere utile sapere quanto lavoro assegnare all'officina esterna o quanti ulteriori macchinari acquistare. Dal punto di vista analitico si tratterebbe di sostituire nel grafo precedentemente realizzato le capacità p_i degli archi (s, L_i) con delle funzioni $p_i(\lambda)$ e le capacità m degli archi (T_j, t) con delle funzioni $m(\mu)$, dove λ e μ quantificano rispettivamente il carico di lavoro da assegnare all'esterno e il numero di macchinari da acquistare. Si tratta a questo punto di determinare valori minimi di λ e μ tali da saturare le capacità degli archi (s, L_i). Problemi di questo tipo vanno sotto il nome di *problemi di flusso massimo parametrico* (cfr. sezione III.3.3). ■

Dato un grafo $G = (N, A)$ e due nodi s e t, si definisce *taglio* (N_s, N_t) una partizione di N in due sottoinsiemi N_s ed N_t tale che $s \in N_s$ e $t \in N_t$. Indichiamo rispettivamente

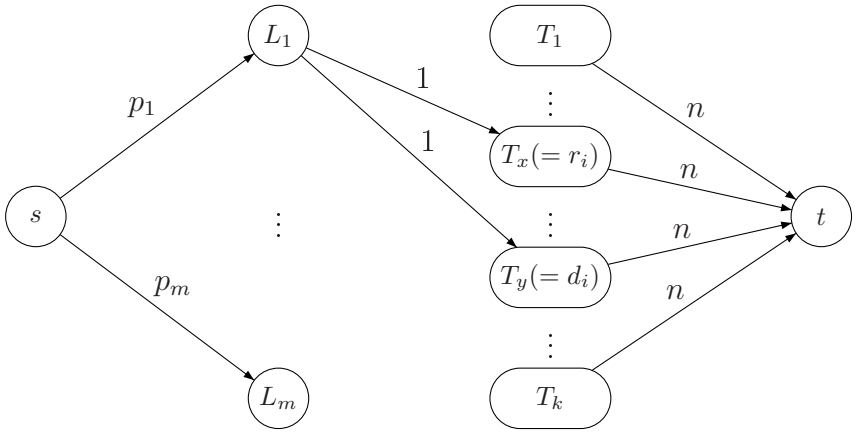


Figura III.2. Grafo di flusso relativo al problema dell'esempio III.7. Per non appesantire il disegno abbiamo omesso gli archi uscenti da L_m, così come abbiamo omesso l'arco fittizio (t, s).

con $A^+_{(N_s,N_t)}$ e $A^-_{(N_s,N_t)}$ gli archi diretti e inversi che attraversano il taglio, ovvero

$$A^+_{(N_s,N_t)} = \{(i,j) \in A |\, i \in N_s, j \in N_t\},$$
$$A^-_{(N_s,N_t)} = \{(j,i) \in A |\, j \in N_t, i \in N_s\},$$

abbreviamo la notazione con A^+ e A^- quando il taglio (N_s, N_t) risulta chiaro dal contesto. Si definisce *capacità* del taglio la quantità

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

Enunciamo senza dimostrazione il seguente risultato.

Teorema III.1 (Flusso Massimo-Taglio Minimo). *Il massimo valore dei flussi ammissibili su $G = (N, A)$ dalla sorgente $s \in N$ alla destinazione $t \in N$ è pari alla minima delle capacità dei tagli (N_s, N_t).* □

Il teorema precedente è di notevole importanza per dimostrare la correttezza degli algoritmi che discuteremo, e può essere riparafrasato dicendo che, dato un flusso ammissibile $\boldsymbol{x}$, ogni volta che esiste un taglio (N_s, N_t) con capacità pari al flusso totale[2] che stiamo spedendo, allora quello è un taglio di capacità minima e il flusso è massimo.

[2]Ricordiamo che il flusso totale spedito da s a t è pari al flusso x_{ts} spedito sull'arco fittizio. Equivalentemente, esso è pari a

$$\sum_{(s,j)\in \mathrm{FS}(s)} x_{sj},$$

o anche a

$$\sum_{(i,t)\in \mathrm{BS}(t)} x_{it}.$$

III.3.2 Algoritmi risolutivi

Esistono diversi algoritmi risolutivi per il problema del flusso massimo, di seguito ne descriveremo alcuni. É utile introdurre la nozione di grafo residuale, che verrà impiegata negli algoritmi che verranno presentati.

Dato un grafo $G = (N, A)$ con capacità sugli archi $\boldsymbol{u}$, e dato un flusso $\boldsymbol{x}$ che rispetti i vincoli di capacità (ma non necessariamente quelli di conservazione del flusso), si definisce *grafo residuale* il grafo $G_R(\boldsymbol{x}) = (N, A_R(\boldsymbol{x}))$, dove

$$\begin{aligned} A_R(\boldsymbol{x}) &= A_R^+(\boldsymbol{x}) \cup A_R^-(\boldsymbol{x}), \\ A_R^+(\boldsymbol{x}) &= \{(i,j) \in A |\, x_{ij} \leq u_{ij}\}, \\ A_R^-(\boldsymbol{x}) &= \{(j,i) |\, (i,j) \in A, x_{ij} \geq 0\}, \end{aligned}$$

dunque $A_R^+(\boldsymbol{x})$ è l'insieme degli archi di G non saturi, mentre $A_R^-(\boldsymbol{x})$ e l'insieme degli archi diretti in verso opposto agli archi di G su cui scorre un flusso. Nel seguito scriveremo semplicemente A_R, A^+ e A^- in luogo di $A_R(\boldsymbol{x})$, $A_R^+(\boldsymbol{x})$ e $A_R^-(\boldsymbol{x})$ quando il flusso $\boldsymbol{x}$ in considerazione risulterà chiaro dal contesto.

Agli archi del grafo residuale è possibile assegnare una *capacità residuale*

$$\boldsymbol{r} = (r_{ij})_{(i,j) \in A_R},$$

definita ponendo

$$r_{ij} = \begin{cases} u_{ij} - x_{ij} & \text{se } (i,j) \in A^+ \text{ (ovvero se } x_{ij} \leq u_{ij}) \\ x_{ji} & \text{se } (i,j) \in A^- \text{ (ovvero se } x_{ij} > 0) \end{cases} \tag{III.3}$$

In sintesi, il grafo residuo contiene al più due "rappresentanti" per ogni arco $(i,j) \in A$: un primo appartenente ad A^+ diretto come (i,j), la cui capacità residua r_{ij} rappresenta quanto flusso può essere ulteriormente spedito su (i,j) nel grafo originale; un secondo arco appartenente ad A^- diretto in verso opposto rispetto a (i,j), la cui capacità residua r_{ij} rappresenta di quanto è possibile diminuire il flusso su (i,j) nel grafo originale.

III.3.2.1 Algoritmi basati sui cammini aumentanti

Questa classe di algoritmi è motivata dal seguente risultato.

Proposizione III.2. *Dato un grafo $G = (N, A)$ con capacità $\boldsymbol{u}$ ed un flusso ammissibile $\boldsymbol{x}$, i seguenti due casi sono mutuamente esclusivi:*

- *esiste un cammino da s a t in $G_R(\boldsymbol{x})$,*
- *esiste un taglio (N_s, N_t) sul grafo residuale $G_R(\boldsymbol{x})$ tale che*[3] $A^+_{(N_s,N_t)} = \varnothing$. □

[3] É importante sottolineare che il taglio è riferito al grafo residuale, dunque $A^+_{(N_s,N_t)} = \varnothing$ significa che non esistono archi del grafo residuale che attraversano il taglio da N_s verso N_t.

Alla luce del teorema III.1, la proposizione precedente afferma che esiste un cammino dalla sorgente alla destinazione sul grafo residuale se e solo se il flusso attuale $\boldsymbol{x}$ non è massimo.

Si definisce *cammino aumentante* un percorso

$$P = [(s, n_1), (n_1, n_2), ..., (n_{k-1}, n_k), (n_k, t)]$$

su $G_R(\boldsymbol{x})$ che porti da s a t. Si dice *capacità del cammino aumentante* P la quantità positiva

$$\theta_P = \min \{r_{ij} | \, (i,j) \in P\} .$$

Trovato un cammino aumentante P è possibile incrementare il flusso totale spedito da s a t, modificando $\boldsymbol{x}$ nel modo seguente:

$$\begin{array}{ll} x_{ij} \leftarrow x_{ij} + \theta_P & \text{se } (i,j) \in A_R^+(\boldsymbol{x}), \\ x_{ji} \leftarrow x_{ji} - \theta_P & \text{se } (i,j) \in A_R^-(\boldsymbol{x}). \end{array} \tag{III.4}$$

É facile verificare che così facendo si continua ad ottenere un flusso ammissibile.

Fatte queste premesse è possibile fornire l'algoritmo CamminiAumentanti per la risoluzione del problema del flusso massimo mediante l'aumento di flusso sui cammini aumentanti. Lo pseudocodice richiama le procedure InviaFlusso e Aggiorna: la prima incrementa il flusso inviato sul cammino di G corrispondente al cammino aumentante P in G_R così come indicato nella (III.4), la seconda ricostruisce il grafo residuale del flusso $\boldsymbol{x}$ appena modificato.

Algoritmo III.3 CamminiAumentanti$(G, \boldsymbol{x})$

```
1: x ← 0
2: Costruisci(G_R(x))
3: while ∃ cammino P da s a t in G_R(x) do
4:     θ_P ← min {r_ij | (i,j) ∈ P}
5:     InviaFlusso(θ, P)
6:     Aggiorna(G_R(x))
```

In realtà l'algoritmo fornito andrebbe completato, in quanto non specifica come fare a determinate il cammino aumentante P (riga 3). Così enunciato, esso è noto in letteratura come *algoritmo di Ford-Fulkerson*. Senza alcuna indicazione sulla scelta di P possono esistere alcuni grafi "patologici" in cui la complessità computazionale della procedura diventa esponenziale. Naturalmente è possibile, specificando la scelta di P, ottenere risultati migliori in termini di complessità: l'*algoritmo di Edmons-Karp* seleziona P come cammino più corto da s a t, ed ha una complessità dell'ordine di $O(m^2 n)$, dove $m = |A|$ e $n = |N|$; altre due versioni dell'algoritmo, note rispettivamente come *algoritmo di Dinič* e *algoritmo dei 3 indiani* hanno invece una complessità dell'ordine di $O(n^2 m)$.

Osservazione III.1. Nel seguito del capitolo, salvo diversa indicazione, utilizzeremo sempre la seguente notazione

$$n = |N| , \qquad m = |A| ,$$

dove (N, A) è il grafo con cui si sta lavorando. □

III.3.2.2 Algoritmo di Capacity Scaling

Rimanendo sempre nell'ambito degli algoritmi basati sui cammini aumentanti, vorremmo un algoritmo che scegliesse cammini aumentanti di capacità residua elevata, in modo da velocizzare il raggiungimento dell'ottimo. Tuttavia, la ricerca su grafo di un cammino a capacità massima è un problema che comporta un calcolo più oneroso rispetto alla scelta di un cammino qualsiasi, dunque dovremo accontentarci di scegliere di volta in volta cammini aumentanti approssimativamente vicini al cammino di capacità residua massima, ma che possano essere ricavati rapidamente. Questa è l'idea che sta alla base dell'*algoritmo di Capacity Scaling*, che ora descriveremo più dettagliatamente.

Dato un grafo $G = (N, A)$ con capacità sugli archi $\boldsymbol{u}$ ed un flusso ammissibile $\boldsymbol{x}$, indichiamo con $G_R(\boldsymbol{x}, \Delta)$ il grafo residuo *filtrato* degli archi di capacità residua minore di Δ: in altre parole si ha $G_R(\boldsymbol{x}, \Delta) = (N, A_R(\boldsymbol{x}, \Delta))$, dove

$$A_R(\boldsymbol{x}, \Delta) = \{(i, j) \in A_R(\boldsymbol{x}) | \, r_{ij} \geq \Delta\} .$$

L'algoritmo parte ponendo Δ pari alla capacità massima degli archi di G, ricercando cammini aumentanti in $G_R(\boldsymbol{x}, \Delta)$, dimezzando Δ ogniqualvolta $G_R(\boldsymbol{x}, \Delta)$ non contenga più alcun cammino aumentante. Mostriamo lo pseudocodice dell'algoritmo CAPACITY-SCALING.

Algoritmo III.4 CAPACITYSCALING$(G, \boldsymbol{x})$

1: $\boldsymbol{x} \leftarrow \boldsymbol{0}$
2: $u_{\max} \leftarrow \max \{u_{ij} | \, (i, j) \in A\}$
3: $\Delta \leftarrow 2^{\lfloor \lg u_{\max} \rfloor}$
4: **while** $\Delta \geq 1$ **do**
5: **while** $\exists$ cammino P da s a t in $G_R(\boldsymbol{x}, \Delta)$ **do**
6: $\theta_P \leftarrow \min \{r_{ij} | \, (i, j) \in P\}$
7: INVIAFLUSSO(θ, P)
8: AGGIORNA$(G_R(\boldsymbol{x}))$
9: $\Delta \leftarrow \Delta / 2$

Si noti che nella riga 3 viene usata l'inusuale espressione

$$2^{\lfloor \lg u_{\max} \rfloor}, \tag{III.5}$$

se $u_{\max}$ è una potenza di 2, allora

$$2^{\lfloor \lg u_{\max} \rfloor} = u_{\max},$$

mentre in generale la (III.5) è la più grande potenza di 2 minore o uguale a $u_{\max}$.

La correttezza dell'algoritmo segue dall'osservazione che $G_R(\boldsymbol{x}, 1) = G_R(\boldsymbol{x})$, quindi all'ultima iterazione l'algoritmo si comporta come un qualsiasi algoritmo di cammini aumentanti.

Analizziamo la complessità della procedura CAPACITYSCALING. Il ciclo while più esterno (righe 4÷11) viene ripetuto $\lfloor \lg u_{\max} \rfloor$ volte, mentre il corpo del ciclo while più

interno (righe 6÷8) ha una complessità dell'ordine di $O(m)$. Resta da determinare un limite superiore sul numero di volte che viene ripetuto il ciclo while più interno (righe 5÷9). Supponiamo che ad un certo istante durante l'esecuzione dell'algoritmo, con $\Delta > 1$, risulti che non esistono cammini aumentanti in $G_R(\boldsymbol{x}, \Delta)$. Allora dalla proposizione III.2 segue che esiste un taglio (N_s, N_t) in $G_R(\boldsymbol{x}, \Delta)$ tale che nessun arco di quest'ultimo lo attraversi da N_s a N_t. Dunque, nel grafo residuale non filtrato $G_R(\boldsymbol{x})$, tutti gli archi che attraversano il taglio da N_s a N_t hanno capacità strettamente minore di Δ, e la capacità complessiva non potrà superare $m\Delta$. A questo punto procediamo dimezzando Δ, ovvero

$$\Delta' = \frac{\Delta}{2},$$

e passando all'iterazione successiva del ciclo while più esterno. Supponiamo che il flusso massimo $\boldsymbol{x}^*$ sia maggiore del flusso $\boldsymbol{x}$ calcolato a questo punto dell'algoritmo. Tuttavia, dal limite sulla capacità del taglio (N_s, N_t) appena discusso, deve risultare

$$\boldsymbol{x}^* - \boldsymbol{x} \leq m\Delta,$$

mentre i cammini aumentanti di $G_R(\boldsymbol{x}, \Delta')$ avranno capacità minima maggiore o uguale a $\Delta' = \Delta/2$: nel caso pessimo serviranno $(m\Delta)/(\Delta/2) = 2m$ aumenti di flusso per raggiungere il flusso massimo $\boldsymbol{x}^*$, dunque il numero di iterazioni del ciclo while più interno sarà limitato da $2m$. In sintesi la complessità dell'algoritmo risulta dell'ordine di $O(2m \cdot m \lg u_{\max}) = O(m^2 \lg u_{\max})$. Si noti che $\lg u_{\max}$ è pari al massimo numero di bit necessari per memorizzare la capacità di un arco, dunque l'algoritmo risulta polinomiale rispetto alle dimensioni dell'input.

III.3.2.3 Limiti degli algoritmi basati sui cammini aumentanti

L'invariante degli algoritmi basati sui cammini aumentanti è che in ogni istante durante l'esecuzione mantengono un flusso $\boldsymbol{x}$ che risulta ammissibile. Se questo ha il vantaggio che in ogni istante possiamo interrompere l'algoritmo e accontentarci della soluzione approssimata finora trovata, per contro ha lo svantaggio che su certi grafi mal congegnati la convergenza verso l'ottimo si rivela estremamente lenta. Prendiamo ad esempio il grafo in figura III.3. Per k molto grande, un algoritmo basato sui cammini aumentanti produrrebbe degli aumenti di flusso di un'unità alla volta, ognuno dei quali comporta l'attraversamento di un numero elevato di archi, raggiungendo così il flusso massimo con notevole lentezza.

III.3.2.4 Algoritmi basati sui preflussi

Per ottenere prestazioni migliori rispetto agli algoritmi basati sui cammini aumentanti, rinunciamo a mantenere un flusso ammissibile in ogni istante durante la ricerca del flusso massimo, rilassando in maniera opportuna i vincoli di conservazione del flusso.

Vale la pena spendere qualche parola per illustrare l'idea della classe di algoritmi che verrà presentata. Consideriamo il grafo di flusso su cui vogliamo lavorare in analogia con una rete di flusso in cui scorre un fluido: tutti i nodi del grafo rappresentano dei serbatoi di fluido di capacità infinita, mentre gli archi della rete rappresentano dei condotti, capaci di

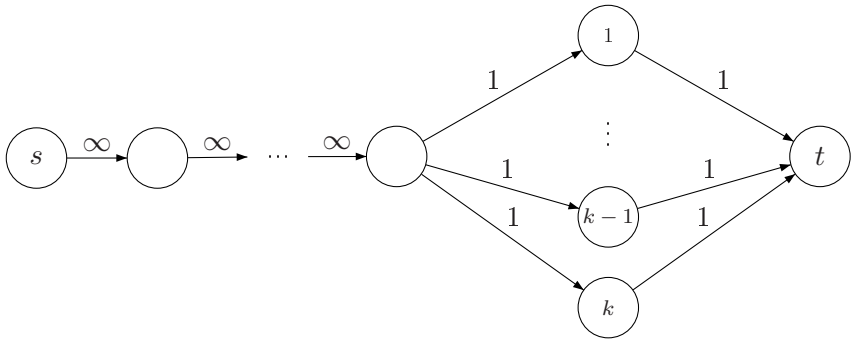


Figura III.3. Esempio di grafo sul quale risulta inefficiente la ricerca del flusso massimo mediante metodi basati sui cammini aumentanti.

una certa portata, nei quali il fluido può scorrere da un serbatoio al successivo. L'obiettivo che vogliamo conseguire è quello di inviare la massima quantità di flusso del fluido dal nodo sorgente s (che "produce" il fluido) al nodo destinazione t (che "consuma" il fluido), in modo che tutti i serbatoi dei nodi intermedi siano vuoti, ovvero non accumulino fluido. Immaginiamo inoltre che ogni nodo della rete sia montato su di una piattaforma, che può essere sollevata in modo da poter spingere (*push*) il fluido verso i nodi adiacenti ad altezza inferiore. In generale il flusso di fluido da un nodo verso un altro nodo più elevato può essere positivo, ma l'operazione di *push* del fluido si effettua sempre da un nodo verso un altro ad altezza inferiore.

L'algoritmo parte impostando tutti i nodi ad altezza 0, ad eccezione del nodo sorgente s che viene posto ad altezza n, dove, lo ricordiamo, indichiamo con n il numero di nodi della rete. Inizialmente inviamo (con una *push*) da s ai nodi adiacenti la massima quantità di flusso spedibile, cioè tale da saturare la capacità degli archi uscenti da s. I nodi adiacenti di s accumuleranno temporaneamente il flusso entrante nel loro serbatoio, ed in seguito lo spediranno verso i nodi a loro adiacenti ad altezza inferiore. Nel momento in cui i soli archi non saturi collegano vertici a pari altezza, vengono innalzati i nodi di partenza di tali archi, in modo da poter spingere il fluido verso i nodi di arrivo degli archi. Iterando la procedura, tutto il fluido spedibile al nodo destinazione t verrà inviato ad esso. Resta solo da regolare il flusso nella rete in modo che il serbatoio di ogni nodo sia vuoto. Per far questo vengono sollevati i nodi interni della rete (potenzialmente ogni nodo diverso da s e t può essere sollevato), in modo da rispedire alla sorgente s il flusso in eccesso accumulato nei vari serbatoi.

Veniamo ora alla formalizzazione dell'algoritmo, cominciando con l'introdurre un po' di notazione. D'ora in avanti indicheremo con $G = (N, A)$ un grafo avente capacità sugli archi $\boldsymbol{u}$, e ci proporremo di risolvere il problema di flusso massimo tra in nodo sorgente $s \in N$ e il nodo destinazione $t \in N$. Un flusso $\boldsymbol{x}$ è detto *preflusso* quando rispetta le

seguenti:

$$0 \leq x_{ij} \leq u_{ij}, \qquad \forall (i,j) \in A, \tag{III.6}$$

$$\sum_{(j,i)\in \mathrm{BS}(i)} x_{ji} - \sum_{(i,j)\in \mathrm{FS}(i)} x_{ij} \geq 0, \qquad \forall i \in N \setminus \{s,t\}\,. \tag{III.7}$$

Dunque nel definire un preflusso abbiamo rilassato il vincolo di conservazione del flusso, limitandoci a richiedere che il flusso entrante in un nodo non superi quello uscente.

Osservazione III.2. Rispetto alla definizione di flusso ammissibile, ora i nodi sorgente s e destinazione t vengono trattati come nodi "privilegiati", in quanto sono "esonerati" dal rispettare la (III.7): così facendo non è più necessario avere un arco fittizio che colleghi t a s. □

Definiamo ora la funzione *eccesso* $e : N \to \mathbb{R}^+$, che associa ad ogni nodo $i \in N$ un valore non negativo, come

$$e(i) := \sum_{(j,i)\in \mathrm{BS}(i)} x_{ji} - \sum_{(i,j)\in \mathrm{FS}(i)} x_{ij}.$$

Si noti che se $\boldsymbol{x}$ è un preflusso la (III.7) impone che $e(i) \geq 0$ per ogni nodo i. Un nodo $i \in N$ è detto *attivo* quando $e(i) > 0$. Un preflusso è un flusso ammissibile se e solo se nessun nodo è attivo, ovvero se e solo se ogni nodo rispetta la (III.7) all'uguaglianza.

Chiamiamo *etichettatura* (o *altezza*) una funzione $h : N \to \mathbb{N}$, che associ ad ogni nodo un numero intero non negativo[4]. Un'etichettatura si dice *compatibile* con il preflusso $\boldsymbol{x}$ quando

$$\begin{aligned} &h(s) = n, \\ &h(t) = 0, \\ &h(i) \leq h(j) + 1 \quad \forall (i,j) \in A_R(\boldsymbol{x}), \end{aligned}$$

dove al solito indichiamo con $G_R(\boldsymbol{x}) = (N, A_R(\boldsymbol{x}))$ il grafo residuale di G rispetto al preflusso $\boldsymbol{x}$. Ricordiamo anche che con $\boldsymbol{r}$ si indica il vettore delle capacità residuali degli archi in $A_R(\boldsymbol{x})$, definite dalla (III.3).

D'ora in poi quando parleremo di un'etichettatura h daremo per scontato che sia compatibile con $\boldsymbol{x}$. Un arco $(i,j) \in A_R(\boldsymbol{x})$ è detto *ammissibile* quando soddisfa le seguenti:

$$\begin{aligned} &h(i) = h(j) + 1, \\ &r_{ij} > 0. \end{aligned}$$

Vista la complicatezza dell'algoritmo, ai fini della sua comprensione, lo spezzeremo in diverse subroutine, per poi fornire la procedura principale PushPreflow. Per migliorare la leggibilità tralasciando i dettagli, assumiamo che ad ogni modifica del preflusso $\boldsymbol{x}$ vengano automaticamente aggiornati il grafo residuale $G_R(\boldsymbol{x})$ e la funzione eccesso $e(\cdot)$.

La prima fase dell'algoritmo consiste nell'assegnare un'etichettatura h a tutti i nodi di G e un preflusso $\boldsymbol{x}$. Procediamo esattamente come descritto nell'introduzione informale discussa a inizio sezione, formalizzata nella procedura Preprocessing.

[4] In letteratura si trova impiegato il simbolo $\mathbb{N}$ per denotare l'insieme dei naturali munito o non munito dello zero. Noi assumeremo la prima, ovvero $\mathbb{N} = \{0, 1, 2, ...\}$.

Algoritmo III.5 PREPROCESSING()

```
1: x ← 0
2: h(s) ← n
3: for all i ∈ N \ {s} do
4:     h(i) ← 0
5: for all (s, i) ∈ FS(s) do
6:     x_si ← u_si
```

Si noti che l'etichettatura prodotta h è compatibile con $\boldsymbol{x}$, infatti i soli lati (i,j) per cui risulta $h(i) > h(j)+1$ sono gli archi uscenti da s (dunque $i = s$), i quali sono stati saturati, dunque non fanno parte di $A_R(\boldsymbol{x})$.

Veniamo alla PUSH, il cui compito è quello di inviare del flusso da un nodo attivo i verso un nodo j tale che $h(i) = h(j)+1$ (i.e. tale che l'invio sia in discesa). Lo scopo della PUSH è quello di diminuire il più possibile $e(i)$ (i.e. di svuotare il più possibile il serbatoio di i) nel rispetto dei vincoli di capacità del preflusso: la quantità δ inviata sarà dunque pari al più piccolo valore tra $e(i)$ e la capacità residua r_{ij} dell'arco (i,j), ovvero

$$\delta = \min(e(i), r_{ij}).$$

Dunque, perché sia eseguibile PUSH(i,j) è necessario che i sia un nodo attivo ($e(i) > 0$), che (i,j) abbia capacità strettamente positiva ($r_{ij} > 0$), e infine che i sia più in alto di j ($h(i) = h(j)+1$).

Se $\delta = r_{ij}$ la push è detta *saturante*, al contrario se $\delta = e(i)$ la push è detta *non saturante*.

Algoritmo III.6 PUSH(i,j)

```
Require: e(i) > 0, r_ij > 0, h(i) = h(j) + 1
1: δ ← min(e(i), r_ij)
2: if (i, j) ∈ A_R^+(x) then
3:     x_ij ← x_ij + δ
4: else if (i, j) ∈ A_R^-(x) then
5:     x_ji ← x_ji − δ
```

Se nel grafo residuale i nodi adiacenti ad un nodo attivo i sono tutti ad altezza maggiore o uguale di $h(i)$, procederemo a rietichettare i (i.e. innalzare la piattaforma di i) in modo che la sua etichetta sia strettamente maggiore della più piccola etichetta dei nodi adiacenti, ovvero:

$$h(i) \leftarrow 1 + \min\{h(j) |\, (i,j) \in A_R(\boldsymbol{x})\}$$

La prossima subroutine è motivata dal seguente risultato.

Lemma III.3. *Su ogni nodo attivo i è possibile applicare* PUSH(i) *oppure rietichettare i.*

Dimostrazione. Per ogni $(i,j) \in A_R(\boldsymbol{x})$ deve risultare $h(i) \leq h(j)+1$, per definizione di etichettatura compatibile con $\boldsymbol{x}$. Se non si può applicare PUSH(i), allora deve essere $h(i) < h(j)+1$, ed essendo $h(i)$ intero risulta $h(i) \leq h(j)$, e dunque i può essere rietichettato. □

A questo punto siamo pronti per fornire la PUSH/RELABEL, che effettua un'operazione tra push e rietichettatura su un nodo attivo i.

Algoritmo III.7 PUSH/RELABEL(i)

Require: $e(i) > 0$
1: **if** $\exists$ arco ammissibile $(i,j) \in A_R(\boldsymbol{x})$ **then**
2: PUSH(i,j)
3: **else**
4: $h(i) \leftarrow 1 + \min\{h(j) | (i,j) \in A_R(\boldsymbol{x})\}$

La procedura principale dell'intero algoritmo, che prende il nome di *algoritmo Push-Preflow*, è la PUSHPREFLOW. Si noti che, così come per l'algoritmo di Ford-Fulkerson, anche qui la procedura non è completamente specificata: a seconda del criterio seguito per scegliere il nodo i nella riga 2, la complessità computazionale dell'algoritmo risulta diversa. Si noti che, invece, la correttezza non dipende da come viene scelto il nodo i.

Algoritmo III.8 PUSHPREFLOW()

1: PREPROCESSING
2: **while** $\exists i \in N \setminus \{s,t\} : e(i) > 0$ **do**
3: PUSH/RELABEL(i)

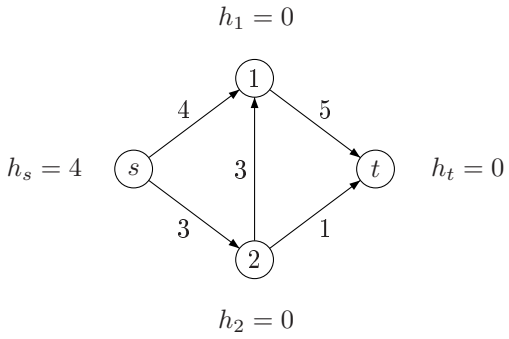


Figura III.4. Grafo di flusso dell'esempio III.8. Accanto ad ogni nodo compare l'etichettatura iniziale, così come viene assegnata dalla PREPROCESSING. Tale procedure assegna anche il flusso iniziale $\boldsymbol{x}$ nullo su tutti gli archi salvo che in quelli uscenti da s, che vengono saturati: $x_{s1} \leftarrow 4$, $x_{s2} \leftarrow 3$.

Esempio III.8. Si consideri il grafo di flusso in figura III.4. La figura III.5 mostra i passi effettuati dall'algoritmo PUSHPREFLOW per trovare il flusso massimo. ■

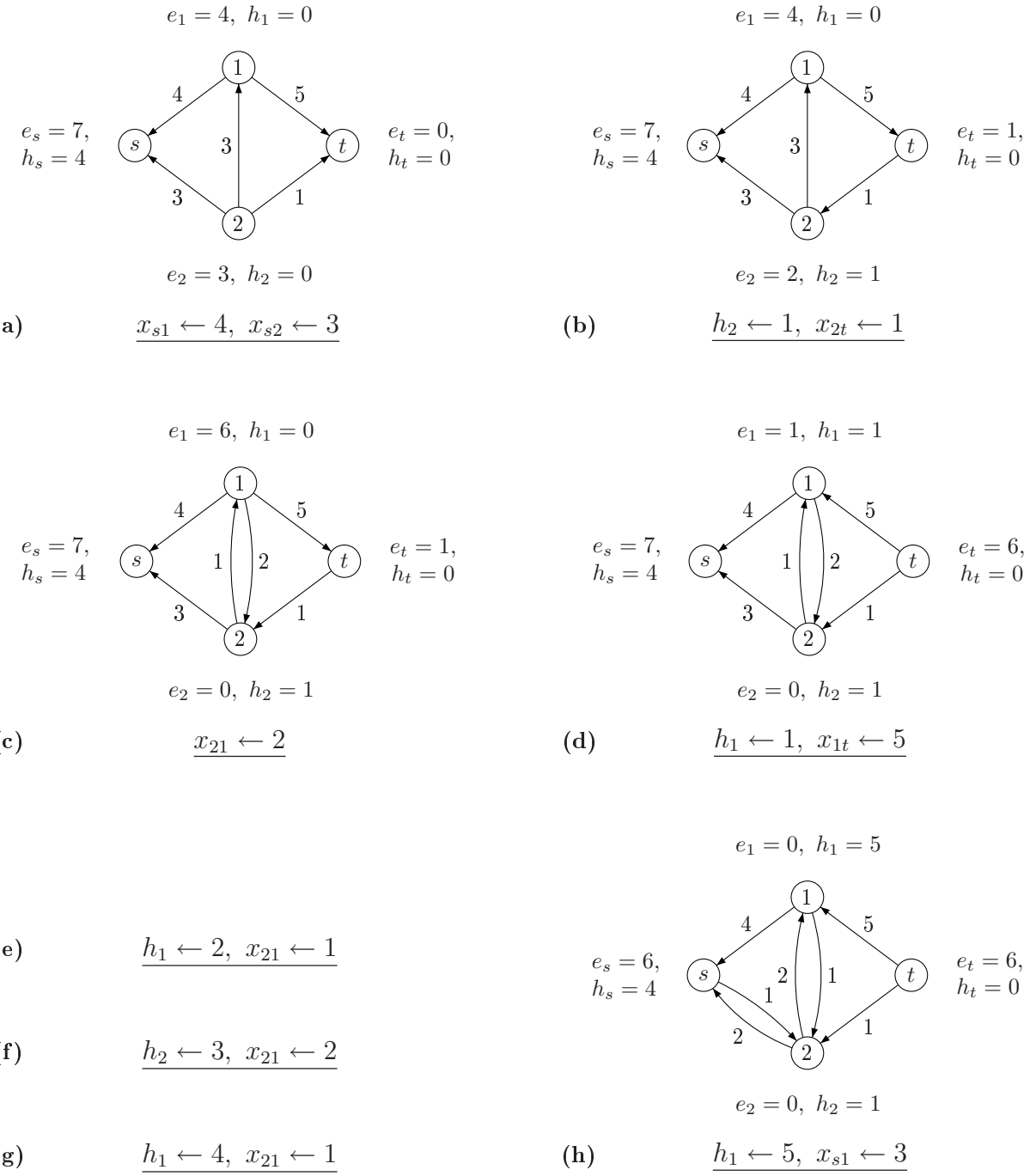


Figura III.5. Passaggi dell'algoritmo PushPreflow sul grafo di figura III.4. I passi **(a)**,...,**(h)** mostrano il grafo residuale corrente, mentre sotto ogni grafo sono mostrate (con sottolineatura) le istruzioni di push o rietichettatura eseguite.

III.3.2.5 Correttezza dell'algoritmo Push-Preflow

In questa sezione dimostreremo che, quando termina, la procedura PushPreflow fornisce un flusso ammissibile massimo. La terminazione dell'algoritmo verrà invece discussa assieme alla sua complessità nelle prossime sezioni.

Per arrivare al risultato che cerchiamo, ci serviremo di alcuni lemmi preliminari.

Lemma III.4. *Se $\boldsymbol{x}$ è un preflusso e h è un'etichettatura compatibile con $\boldsymbol{x}$, allora non esiste alcun cammino da s a t nel grafo residuale $G_R(\boldsymbol{x})$.*

Dimostrazione. Dalla definizione di etichettatura compatibile risulta

$$h(i) \leq h(j) + 1 \quad \forall (i,j) \in A_R(\boldsymbol{x}),$$

unito ai vincoli $h(s) = n$ e $h(t) = 0$, quindi un qualsiasi percorso da s a t in $G_R(\boldsymbol{x})$ sarà costituito da almeno n archi. Inoltre in un grafo esiste un percorso tra due nodi se e solo se esiste un percorso semplice (ovvero senza nodi ripetuti) tra quei due nodi, ed un percorso semplice in un grafo con n nodi può essere lungo al più $n-1$ nodi. Dunque non esiste un percorso semplice da s a t in $G_R(\boldsymbol{x})$, e nemmeno un percorso generale. □

Lemma III.5. *Durante l'esecuzione di* PushPreflow, *per ogni nodo $i \in N$, l'altezza $h(i)$ non decresce mai, e quando cresce aumenta di almeno una unità.*

Dimostrazione. $h(i)$ cambia solo quando il vertice i viene rietichettato, il che avviene solo se $h(i) \leq h(j)$ per ogni $j \in A_R(\boldsymbol{x})$, e $h(i)$ viene posta a $1 + \min\{h(j) | (i,j) \in A_R(\boldsymbol{x})\}$, dunque cresce di almeno 1. □

Lemma III.6. *Durante l'esecuzione di* PushPreflow, *la funzione h è un'etichettatura compatibile con $\boldsymbol{x}$.*

Dimostrazione. Al termine del preprocessing h è un'etichettatura compatibile con $\boldsymbol{x}$. Resta da dimostrare che rimane tale quando varia $\boldsymbol{x}$ oppure h.

Consideriamo la rietichettatura di un nodo i. Per ogni $(i,j) \in A_R(\boldsymbol{x})$ al termine della rietichettatura risulta $h(i) \leq h(j) + 1$, mentre per ogni $(k,i) \in A_R(\boldsymbol{x})$ risulta $h(k) \leq h(i) + 1$, ed essendo che $h(i)$ cresce di almeno 1 (cfr. lemma III.5), dopo la rietichettatura risulta $h(k) < h(i) + 1$. Dunque la rietichettatura mantiene la compatibilità.

Per quanto riguarda la push su un nodo i, se è saturante fa sparire l'arco interessato da $A_R(\boldsymbol{x})$. Se non è saturante potrebbe aggiungere l'arco (i,j) a $A_R(\boldsymbol{x})$, ma per poter essere eseguita deve rispettare $h(i) = h(j) + 1$, dunque $h(j) = h(i) - 1 < h(i) + 1$. □

Il risultato che cerchiamo è il seguente.

Teorema III.7. *Alla terminazione, l'algoritmo* PushPreflow *fornisce un flusso ammissibile massimo.*

Dimostrazione. É facile verificare che per tutta la durata dell'algoritmo PushPreflow il flusso $\boldsymbol{x}$ è un preflusso, infatti lo è al termine del preprocessing e continua ad esserlo a seguito delle push. Alla terminazione tutti i nodi hanno eccesso nullo, quindi il preflusso $\boldsymbol{x}$ rispetta i vincoli di conservazione, e dunque è un flusso ammissibile. Essendo che, per il lemma III.6, h è un'etichettatura compatibile con $\boldsymbol{x}$, per il lemma III.4 ne consegue che non esiste alcun percorso da s a t in $G_R(\boldsymbol{x})$. Allora per il Teorema del Flusso Massimo-Taglio Minimo (teorema III.1) $\boldsymbol{x}$ è un flusso massimo ammissibile. □

III.3.2.6 Analisi di complessità dell'algoritmo Push-Preflow

Analizziamo ora la complessità dell'algoritmo PushPreflow. Per valutarla cerchiamo di determinare quante volte vengono eseguite le tre operazioni di rietichettatura e di push, saturante e non saturante. Per facilitare l'analisi ci serviamo del seguente lemma.

Lemma III.8. *Consideriamo un grafo $G = (N, E)$ con sorgente s e sia $\boldsymbol{x}$ un preflusso. Allora, per ogni nodo attivo i esiste un cammino semplice da s a i nel grafo residuale $G_R(\boldsymbol{x})$.*

Dimostrazione. Per ogni nodo attivo i, sia $I = \{j : \exists \text{ un cammino da } i \text{ a } j \text{ in } G_R(\boldsymbol{x})\}$ e supponiamo per assurdo che $s \notin I$. Sia inoltre $\bar{I} = N \setminus I$. Per ogni coppia di vertici $k \in \bar{I}$ e $j \in I$ vale che $\boldsymbol{x}_{kj} \leq 0$. Infatti, se fosse $\boldsymbol{x}_{kj} > 0$, avremmo $\boldsymbol{x}_{jk} < 0$ e dunque $r_{jk} = u_{jk} - \boldsymbol{x}_{jk} > 0$. Quindi esisterebbe un arco $(j, k) \in G_R(\boldsymbol{x})$ e un cammino da i a k passante per j in contraddizione con l'ipotesi che $k \in \bar{I}$. Da ciò deriva che $\boldsymbol{x}_{\bar{I}I} \leq 0$, poichè ogni termine della sommatoria è negativo o nullo e quindi

$$e(I) = \boldsymbol{x}_{NI} = \boldsymbol{x}_{\bar{I}I} + \boldsymbol{x}_{II} = \boldsymbol{x}_{\bar{I}I} \leq 0.$$

Poichè gli eccessi sono per definizione non negativi, segue $e(i) = 0$ per ogni $i \in I \subseteq N \setminus \{s\}$. In particolare abbiamo $e(i) = 0$, in contraddizione con l'ipotesi che i è un nodo attivo. □

Ora cerchiamo un bound anche per le altezze dei vertici e per il numero di rietichettature effettuate dall'algoritmo nel corso della sua esecuzione.

Lemma III.9. *Durante l'esecuzione di* PushPreflow *su $G(N, E)$ risulta $h(i) \leq 2n-1$, con $n = |N|$.*

Dimostrazione. Per i nodi sorgente e destinazione la relazione è verificata in conseguenza dell'etichettatura compatibile con il preflusso. Consideriamo i restanti vertici. L'etichetta del nodo i cambia quando viene eseguita Relabel(i), che ha come requisito $e(i) > 0$. Per il lemma III.8 esiste un cammino semplice p da i alla sorgente s in $G_R(\boldsymbol{x})$. Il cammino è composto dai nodi $v_0, v_1, \ldots, v_k$ con $k \leq n - 1$. Dal lemma III.4 abbiamo che $h(v_i) \leq h(v_{i-1}) + 1$ ed espandendo le disuguaglianze lungo il cammino p otteniamo

$$h(i) = h(v_0) \leq h(v_k) + k \leq h(s) + (n - 1) = 2n - 1.$$ □

Come conseguenza del lemma III.9 abbiamo

Lemma III.10. *Durante l'esecuzione di* PushPreflow *su $G(N, E)$ con $n = |N|$, il numero di operazioni di rietichettatura è al massimo $2n - 1$ per ogni vertice e in totale inferiore a $2n^2$.*

Dimostrazione. Come detto in precedenza, le rietichettature possono essere effettuate per i nodi $i \in N \setminus \{s, t\}$. La Relabel($i$) incrementa $h(i)$. Inizialmente $h(i) = 0$ e cresce al massimo fino a $2n - 1$. Quindi ogni vertice viene rietichettato al più $2n - 1$ volte e in totale abbiamo $(2n - 1)(n - 2) \leq 2n^2$ operazioni di Relabel(i). □

Per quanto riguarda le push abbiamo

Lemma III.11. *Il numero di push saturanti è $O(nm)$.*

Dimostrazione. Supponiamo di aver eseguito PUSH(i,j) (contiamo le push saturanti da i a j insieme a quelle da j a i). Nel grafo residuale non avremo più l'arco (i,j) ma solo quello (j,i). Avremo $h(j) = h(i) - 1$. Affinchè avvenga un'altra PUSH(i,j) l'algortimo deve prima spedire flusso da j a i ma questo non può accadere perchè (j,i) non è ammissibile: perchè lo diventi $h(j)$ deve aumentare di almeno 2, visto che $h(i)$ non diminuisce mai durante l'esecuzione. Anche $h(i)$ deve crescere di almeno 2 durante le push saturanti tra j e i. Ma dal lemma III.9 sappiamo che $h(k) \leq 2n-1$ e dunque il numero di volte che l'altezza di un vertice può crescere di 2 è inferiore a n. Siccome almeno una tra $h(i)$ e $h(j)$ deve aumentare di 2 tra le push saturanti tra i e j, il numero di push saturanti per ogni vertice risulta inferiore a $2n$. In totale, considerando tutti gli archi del grafo ($|E| = m$), abbiamo un bound superiore di $2nm$. □

Lemma III.12. *Il numero di push non saturanti è $O(n^2m)$.*

Dimostrazione. Utilizziamo le tecniche di valutazione della complessità ammortizzata viste in precedenza, in particolare il *metodo del potenziale* (cfr. sezione III.2.3). Definiamo la funzione potenziale

$$\Phi(\boldsymbol{x}, h) = \sum_{i:\, e(i)>0} h(i)$$

Vediamo ora l'effetto delle tre operazioni possibili sulla funzione potenziale. Quando si effettua una RELABEL(i), $\Phi(\boldsymbol{x}, h)$ cresce di almeno una unità, dal lemma III.9 sappiamo che $h(i) \leq 2n-1$, dunque globalmente il contributo è di $(2n-1)n$, ordine $O(n^2)$. Anche la push saturante incrementa $\Phi(\boldsymbol{x}, h)$. Supponiamo di avere l'arco (i,j) in $G_R(\boldsymbol{x})$. Per effettuare la push saturante deve essere $e(i) > 0$. Dopo l'esecuzione di PUSH(i,j), $e(i)$ si mantiene positiva e non aggiunge contributo, mentre $e(j)$ aumenta di una certa quantità δ. Nel caso pessimo questo incremento sarà pari a $2n-1$ e quindi moltiplicato per il numero di push saturanti dà un contributo globale di $(2n-1)(2nm)$ pari a $O(n^2m)$. Dimostriamo ora che una push non saturante da i a j fa diminuire la funzione potenziale di almeno una unità. Prima della push non saturante $e(i) > 0$, mentre non abbiamo vincoli per $e(j)$. Dopo la push i non è più attivo, mentre j lo deve essere, almeno che non sia la sorgente della rete. Quindi $\Phi(\boldsymbol{x}, h)$ diminuisce di esattamente $h(i)$ unità ed aumenta di 0 o $h(j)$. Essendo $h(i) = h(j) - 1$, globalmente abbiamo

$$\Phi(\boldsymbol{x}, h) = \Phi(\boldsymbol{x}, h) - h(i) + h(j) = \Phi(\boldsymbol{x}, h) - 1.$$

□

Teorema III.13. *La complessità di* PUSHPREFLOW *è $O(n^2m)$.*

Dimostrazione. Nel corso dell'algoritmo gli incrementi sono nell'ordine di $O(n + n^2m)$ e i decrementi nell'ordine di $O(n^2m)$. La complessità di PUSHPREFLOW è dunque $O(n^2m)$ □

Dall'analisi , la complessità di PUSHPREFLOW risulta migliore rispetto a quella dell'*algoritmo di Edmons-Karp*, e questo senza aver definito in alcun modo la scelta dell'arco ammissibile. *Goldberg e Tarjan* proposero un algoritmo di complessità dell'ordine di $O(n^3)$. Alla base dell'algoritmo da loro proposto vi sono le seguenti modifiche:

- La scelta dell'arco ammissibile in PUSH/RELABEL avviene con una scansione ciclica, in cui si tiene traccia dell'ultimo arco non saturo che si è scelto in precedenza, evitando di riconsiderare archi saturi.

- In PushPreflow si sceglie sempre il nodo di altezza massima. Questo richiede l'uso di particolari strutture dati di semplice implementazione, che portano la complessità a $O(n^2\sqrt{m})$. Dalle recenti analisi sperimentali questa variante risulta tra le più efficienti in pratica.

III.3.2.7 Trucchi implementativi per gli algoritmi di Push-Preflow

Se si implementa in pratica l'algoritmo si possono utilizzare alcuni "trucchi". L'altezza $h(i)$ rappresenta per ogni nodo una stima per difetto della lunghezza del cammino da i al nodo terminale t se $h(i) < n$, oppure da i al nodo sorgente, se $h(i) > n$. Si può migliorare l'algoritmo facendo di tanto in tanto una visita nel grafo residuale, valutando esattamente le distanze: questo provoca un aggravio dell'ordine di $O(m)$. In particolare si effettua una rietichettatura esatta quando $h(i) = n$, in modo da individuare prima possibile il taglio (N_s, N_t), che rappresenta il limite per il nodo i, che non può più mandare flusso verso t. Questa operazione ha un costo di $O(m)$. Un ulteriore trucco consiste nel mettere in scala le etichette associate con il loro nodo, ordinandole da 0 a $2n-1$. Per rispettare la compatibilità delle altezze con il preflusso, l'ordinamento tra le $h(i)$ deve essere continuo, non ci può essere un salto tra l'altezza $h(k)$ e $h(k)+1$. Nel momento in cui si verifica questa condizione, di fatto si è determinato un taglio e si possono spostare le etichette (e quindi i nodi) in fondo alla lista perchè non esisterà un cammino nel grafo residuale.

III.3.3 Flusso massimo parametrico

Il problema del flusso massimo parametrico rappresenta una generalizzazione del problema del flusso massimo, in cui le capacità degli archi non hanno un valore costante, bensì sono funzioni lineari di un parametro λ, pertanto anche il flusso risulterà funzione di λ. Vediamo un esempio per illustrare meglio il problema.

Esempio III.9 (Rete di telecomunicazioni). Consideriamo una rete di telecomunicazioni, descritta da un grafo $G(N, E)$, e immaginiamo di dover testare ogni arco della rete periodicamente per verificare il suo funzionamento. Associamo quindi ad ogni arco un valore α_{ij} che rappresenta il numero di test da effettuare sull'arco (i, j). Il test può essere effettuato da ogni nodo estremo del link, pertanto si può associare un valore β_i che dà una misura della potenza di calcolo del nodo, cioè del numero di test che i può eseguire in una unità di tempo. Immaginiamo di voler sapere se in una unità di tempo riusciamo ad effettuare tutti i test. Possiamo ricondurci ad un problema di flusso massimo su un grafo $\widehat{G}(\widehat{N}, \widehat{E})$ bipartito. La struttura del grafo è la seguente

- Aggiungiamo un nodo in $\widehat{N}$ per ogni nodo $c \in E$.
- $\widehat{N} = N \cup \{n_{ij} : (i, j) \in E\}$. In questo modo individuiamo due classi N_1 e N_2 che rappresentano rispettivamente i nodi originari della rete e i nuovi nodi derivati dai link esistenti nella rete.
- Aggiungiamo un nodo sorgente s e un nodo destinazione t.
- Colleghiamo il nodo s con tutti i nodi di N_1
- Colleghiamo ogni nodo $i \in N_1$ con i nodi $n_{ij} \in N_2$.

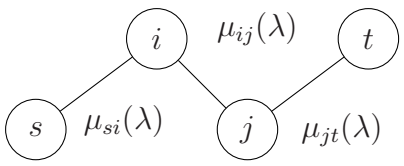


Figura III.6. Grafo della rete di telecomunicazioni con capacità parametriche.

- Ogni arco (s, i) con $i \in N_1$ ha come capacità la potenza di calcolo del nodo i.
- Ogni arco (j, t) con $j \in N_2$ ha come capacità α_{ij}.
- Gli archi (i, j) hanno capacità infinita.

Costruito il grafo $\widehat{G}$, il test è effettuabile in una unità di tempo se il flusso massimo V è pari a

$$\sum_{(i,j)\in\widehat{E}} \alpha_{ij}.$$

Se V risulta inferiore a questo valore vuol dire che il test non è effettuabile in una unità di tempo. Si può provare a vedere se è effettuabile in 2 unità, raddoppiando le capacità sugli archi (s, i) e risolvendo nuovamente il problema di flusso massimo. In generale si possono assegnare delle capacità parametriche $\lambda\beta$ e cercare il minimo λ tale che

$$V(\lambda) = \sum_{(i,j)\in\widehat{E}} \alpha_{ij}.$$

■

In generale possiamo definire il problema come ricerca del flusso massimo in un grafo $G(N, A)$ con $s, t \in N$ e capacità

$$u_{ij}(\lambda) = \begin{cases} u_{ij} & (i,j) : i \neq so j \neq t \\ u_{si}(\lambda) & \text{non decrescente} \\ u_{jt}(\lambda) & \text{non crescente} \end{cases}$$

Nell'esempio III.9 alcuni archi hanno delle capacità variabili linearmente. Consideriamo una sequenza di valori $\lambda_0, \lambda_1, \ldots, \lambda_l$. Un semplice algoritmo consiste nel calcolare il flusso massimo con l'algoritmo PushPreflow per ogni λ_i, e con una ricerca binaria trovare il valore ottimo. La complessità è dell'ordine di $O(ln^2m)$ ($O(ln^3)$ per la variante di Goldberg-Tarjan). Tuttavia, è possibile realizzare un algoritmo più efficiente.

Possiamo osservare come nell'esempio della rete di telecomunicazioni (figura III.6) le capacità $\mu_{si}(\lambda)$ sono non decrescenti in λ ($\mu_{si}(\lambda) = \lambda\beta_i$), le capacità $\mu_{jt}(\lambda)$ sono non crescenti in $\lambda(\mu_{jt}(\lambda) = \alpha_j)$, mentre le capacità $\mu_{ij}(\lambda)$ non dipendono da λ. Vorremmo sfruttare questa osservazione ed evitare di dover ricalcolare il flusso l volte, per ogni λ_i. Se consideriamo il flusso x^{λ_k} e il flusso $x^{\lambda_{k+1}}$ con $\lambda_{k+1} > \lambda_k$, vediamo che, al variare di λ, le capacità degli archi (s, i) aumentano, mentre quelle degli archi (j, t) diminuiscono (entrambe possono anche rimanere invariate). Pertanto, anzichè calcolare il flusso massimo ogni volta, possiamo ottenere il nuovo valore del flusso nel seguente modo

$$x_{ij}^{\lambda_{k+1}} = \begin{cases} max\{x_{ij}^{\lambda_k}, \mu_{ij}(\lambda_{k+1})\} & \text{se } i = s \\ min\{x_{ij}^{\lambda_k}, \mu_{ij}(\lambda_{k+1})\} & \text{se } j = t \\ x_{ij}^{\lambda_k} & \text{se } i \neq s \text{ e } j \neq t \end{cases} \tag{III.8}$$

Così calcolato, $\boldsymbol{x}_{ij}$ non è un flusso, in quanto, pur rispettando i vincoli di capacità, non rispetta i vincoli di conservazione dei nodi interni. In ogni nodo interno, infatti, il flusso entrante è superiore al flusso uscente. Da questo segue che $\boldsymbol{x}_{ij}$ è un preflusso, in base alle equazioni (III.6) e (III.7). Potremmo quindi applicare l'algoritmo PushPreflow per λ_1 e in seguito aggiornare i flussi con la regola (III.8). Dobbiamo solo verificare che le etichettature siano compatibili, per poter applicare l'algoritmo. Le etichette sono compatibili se

$$h(i) \leq h(j) + 1 \quad \forall (i,j) \in E_R(\boldsymbol{x}).$$

Se lo sono prima di applicare PushPreflow, lo sono anche successivamente, infatti con i nuovi assegnamenti potremmo solo saturare degli archi (j,t), ma questo non cambierebbe le altezze. La procedura base dell'algoritmo diventa la seguente.

Algoritmo III.9 MaxFlow$(\lambda_1, ..., \lambda_l)$

```
1: x^{λ1} =PushPreflow
2: for all k = 2, ..., l do
3:    if (i ≠ s) ∧ (j ≠ t) then
4:       x_ij^{λi} ← x_ij^{λi−1}
5:    else if i = s then
6:       x_ij^{λi} ← max{x_ij^{λi−1}, μ_ij(λ_i)}
7:    else if j = t then
8:       x_ij^{λi} ← min{x_ij^{λi−1}, μ_ij(λ_i)}
9:    while ∃i : e_i > 0 do
10:      PushRelabel(i)
```

Vediamo ora di analizzarne la complessità. La prima chiamata della procedura PushPreflow, ha un costo di $O(n^2m)$ (o $O(n^3)$), come visto in precedenza. La parte centrale dell'algoritmo, costituita dagli assegnamenti del flusso, ha invece un costo di $O(n)$ ad ogni iterazione. Analizziamo le successive chiamate di PushPreflow all'interno del ciclo *for*. Il costo delle Relabel è già incluso nella prima chiamata, infatti le etichette non devono essere riconteggiate ad ogni iterazione, e lo stesso vale per le push saturanti, in quanto gli archi eliminati nel grafo residuale alla prima iterazione, non sono più presenti alle successive. Le uniche operazioni che dobbiamo considerare sono le push non saturanti. Si può creare un eccesso in ogni nodo (esclusi s e t) e le etichette possono aumentare fino al valore $2n-1$. Pertanto il contributo delle push non saturanti è $O(n^2)$. Complessivamente l'algoritmo ha complessità $O(l(n+n^2)+n^2m)$ (o $O(l(n+n^2)+n^3)$). Vogliamo ora cercare un modo efficace di trovare la sequenza dei λ, che ci permetta di non effettuare tutte le l iterazioni. Dal teorema III.1 sappiamo che il flusso massimo è equivalente al taglio di capacità minima. Consideriamo perciò una sequenza di tagli $({N_s}^1, {N_t}^1), \ldots, ({N_s}^l, {N_t}^l)$

crescenti. Visto che le capacità (s,i) non possono diminuire e che le capacità (j,t) non possono aumentare è evidente che gli insiemi N_s^i tendono ad aumentare. Si può dimostrare il seguente.

Teorema III.14. *Data una sequenza crescente di tagli* $({N_s}^1, {N_t}^1), \ldots, ({N_s}^l, {N_t}^l)$*, risulta*

$$ {N_s}^1 \subseteq {N_s}^2 \subseteq \ldots \subseteq {N_s}^l. $$

Idea di dimostrazione. Sappiamo che $h(i) \geq n$ se $i \in N_s$, mentre $h(i) < n$ se $i \in N_t$. Durante l'algoritmo, grazie alle rietichettature, potranno aumentare delle etichette già in N_s e potrebbero aggiungersene alcune di N_t. Pertanto non è possibile che la capacità del taglio si riduca ma necessariamente dovrà aumentare o al più rimanere costante. Questa proprietà è detta *nesting property.* □

Possiamo scrivere una relazione lineare per il flusso $\boldsymbol{x}(\lambda)$:

$$ \boldsymbol{x}(\lambda) = \mu + \mu(\lambda), $$

dove con μ indichiamo il contributo degli archi interni che non dipende da λ e con $\mu(\lambda)$ quello degli archi (s,i) e (j,t). L'obiettivo diventa quello di determinare il parametro λ_k che rende minimo $\boldsymbol{x}(\lambda)$. Calcoliamo il taglio ${N_s}^1, {N_t}^1$ al variare dei coefficienti λ_i. Il flusso ottenuto è lineare in λ e può essere rappresentato con una retta nel piano. Calcoliamo ora il taglio ${N_s}^l, {N_t}^l$: l'intersezione delle due rette ottenute ci fornisce una prima stima del λ ottimo che cerchiamo. In generale otteniamo una funzione lineare a tratti di cui vogliamo calcolare il massimo. Se abbiamo $n-1$ tratti abbiamo al più $n-2$ breakpoint. Possiamo quindi utilizzare un metodo iterativo per la ricerca del massimo, che procede valutando le intersezioni tra le rette.

Vediamo ora un esempio di applicazione del problema di flusso massimo in cui il grafo ha dimensioni molto elevate (numero dei nodi nell'ordine del milione).

Esempio III.10 (Separazione elementi di una immagine). Immaginiamo di avere una fotografia e di voler separare gli elementi in primo piano da quelli sullo sfondo, ad esempio perchè vogliamo ritoccare i volti degli individui della foto (per esempio togliere i punti rossi degli occhi). Se costruiamo una griglia e andiamo a considerare i singoli pixel (in totale sono P) che costituiscono l'immagine, possiamo utilizzare un algoritmo molto semplice per effettuare l'operazione. Supponiamo di disporre di due coefficienti a_i e b_i che prendono valori elevati, rispettivamente se il pixel i fa parte del soggetto dell'immagine o se il pixel i fa parte dello sfondo. Un algoritmo di separazione, di complessità $O(P)$, è l'algoritmo SeparazionePixel. Questo algoritmo non fa

Algoritmo III.10 SeparazionePixel()

```
1: for all i = 1, ..., P do
2:    if a_i < b_i then
3:       i ∈ SFONDO
4:    else
5:       i ∈ SOGGETTO
```

però una distinzione precisa dei pixel, quindi potrebbe capitare di assegnare un pixel all'insieme sbagliato e perciò, ad esempio, di ritrovarsi con una foto imperfetta a seguito dell'elaborazione. Per raffinare l'algoritmo teniamo conto della relazione di vicinanza tra pixel, supponendo di disporre di un coefficiente p_{ij} per ogni coppia di pixel adiacenti i e j che indichi una penalità in caso di separazione dei due pixel.

Definiti

$$A = \{\text{insieme dei pixel } i \text{ del SOGGETTO}\},$$
$$B = \{\text{insieme dei pixel } i \text{ dello SFONDO}\},$$

possiamo formulare il problema in termini di programmazione lineare come ricerca di un assegnamento mutuamente esclusivo dei pixel al soggetto (insieme A) oppure allo sfondo (insieme B), in modo da massimizzare la funzione $q(A, B)$ così definita

$$q(A,B) = \sum_{i \in A} a_i + \sum_{i \in B} b_i - \sum_{(i,j) \in E:\, i \in A, j \in B} p_{ij}.$$

La funzione $q(A, B)$ è somma di termini positivi e negativi e noi invece vorremmo ricondurci ad un problema di minimo con una funzione composta da termini tutti positivi. Definito

$$Q = \sum_{i=1}^{P} (a_i + b_i),$$

risulta

$$q(A,B) = Q - \sum_{i \in B} a_i - \sum_{i \in A} b_i - \sum_{(i,j) \in E:\, i \in A, j \in B} p_{ij} = Q - q'(A,B),$$

in cui Q è costante. Pertanto massimizzare $q(A, B)$ equivale a minimizzare la funzione a termini positivi

$$q'(A,B) = \sum_{i \in B} a_i + \sum_{i \in A} b_i + \sum_{(i,j) \in E:\, i \in A, j \in B} p_{ij}.$$

A questo punto risulta naturale costruire un grafo non orientato $G(N, E)$ nel seguente modo:

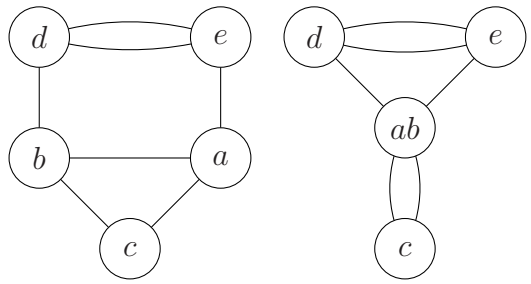
- l'insieme dei nodi N contiene un nodo per ogni pixel dell'immagine, più due nodi speciali s e t:
$$N = \{1, ..., P\} \cup \{s, t\},$$
- l'insieme degli archi contiene un arco (i, j) per ogni coppia di pixel adiacenti i e j, inoltre è completato aggiungendo due archi (s, i) e (i, t) per ogni pixel i:
$$E = \{(i,j) |\, i \text{ adiacente } j\} \cup \{(s,i), (i,t) |\, i \in \text{PIXEL}\}.$$

Assegnamo un vettore capacità $\boldsymbol{u}$ agli archi del grafo, tale che:

$$u_{si} = a_i,\ u_{it} = b_i, \qquad \forall i \in N \setminus \{s,t\},$$
$$u_{ij} = p_{ij}, \qquad \forall (i,j) \in E :\ i,j \in N \setminus \{s,t\}.$$

Costruito il grafo di flusso G non resta che cercare un taglio (N_s, N_t) di capacità minima: trovato il taglio, è facile verificare che assegnando

$$A = N_t \setminus \{t\},$$
$$B = N_s \setminus \{s\},$$



(a) Multigrafo (b) Contrazione arco

Figura III.7. Esempio di contrazione di un arco su multigrafo.

si minimizza $q'(A, B)$. Ricordiamo che, dal teorema di Flusso Massimo-Taglio Minimo (teorema III.1), la capacità del taglio di capacità minima è pari al flusso massimo spedibile da s a t, e una volta trovato il flusso massimo $\boldsymbol{x}$ è possibile individuare il taglio di capacità minima visitando il grafo residuale $G_R(\boldsymbol{x})$ (cfr. teorema III.2). Ci siamo ricondotti, quindi, ad un problema di flusso massimo su un grafo di grandi dimensioni. ■

III.3.3.1 Taglio minimo randomizzato

Quello visto nell'esempio è un tipico problema di separazione. Consideriamo, come caso più generale, un multigrafo $G(V, E)$ connesso. Dato il taglio $C \subseteq E$, il grafo risultante $\widehat{G}(V, E \setminus C)$ è non connesso. Il problema consiste nel trovare il taglio minimo.

Un algoritmo poco efficiente potrebbe cercare il taglio minimo per ogni coppia di nodi, utilizzando ogni volta un algoritmo di flusso massimo. Otterremmo una complessità di $O(n^2 n^3) = O(n^5)$. L'algoritmo proposto da Gomory-Hu riesce a trovare il taglio minimo su un grafo con una complessità di $O(n^4)$.

Si definisce *contrazione* di un arco $(x, y) \in E$ l'operazione che elimina l'arco dal grafo e compatta i due nodi precedentemente collegati restituendo il grafo contratto[5] $G/(x, y)$. Nel caso di un multigrafo la contrazione di un arco prevede l'eliminazione di tutti gli archi tra i due nodi estremi.

Nella figura III.7 si mostra la contrazione di un arco. Dal grafo iniziale (Figura III.7(a)), si contrae l'arco (a, b) ottenendo il nuovo grafo (Figura III.7(b)), in cui i nodi a e b sono compattati nel supernodo ab ed ogni arco (a, v) o (b, v) viene sostituito da un arco (ab, v). Supponendo che K sia il taglio di costo minimo che divide in due G, se contraiamo un arco per parte, il grafo che si ottiene è ancora dello stesso tipo. Quindi, in generale,

[5] In algebra l'operazione di contrazione su grafi è frequente, e viene definita in termini più generali rispetto che nel nostro contesto. In breve, dato il grafo $G = (N, A)$, presi $x, y \in N$, e detta R la più piccola relazione d'equivalenza su N contenente (x, y), si definisce il *grafo quoziente* $G/R = (\tilde{N}, \tilde{A})$, dove

$$\tilde{N} = (N \setminus \{x, y\}) \cup \{xy\}\,,$$
$$\tilde{A} = (A \setminus (\mathrm{BS}(x) \cup \mathrm{BS}(y))) \cup \{(n, xy) |\, (n, x) \in A \text{ oppure } (n, y) \in A\}\,.$$

basterebbe essere in grado di misurare con che probabilità l'arco che si sta contraendo non appartiene al taglio. Si può osservare che se si contrae un arco che non appartiene a K, il taglio rimane minimo anche dopo la contrazione. Quindi un algoritmo randomizzato può essere semplicemente un loop in cui ad ogni iterazione si contrae un arco scelto a caso. L'algoritmo termina quando ci si è ridotti ad un grafo con due nodi ed è corretto se non sono stati contratti archi appartenenti al taglio. Il taglio minimo sarà costituito dagli archi tra i due nodi.

Algoritmo III.11 MinCut()

```
1: H ← G
2: while H ha più di 2 nodi do
3:     scegli (x, y) ∈ E
4:     H ← H(x, y)
```

L'idea dell'algoritmo è quella fornita dalla procedura MinCut. Dobbiamo dunque analizzare con che probabilità la soluzione che otteniamo è quella ottima. Se definiamo come $k = |K|$ la capacità del taglio minimo, ogni nodo di G dovrà avere almeno k archi incidenti, altrimenti K non sarebbe il taglio minimo. Da questo si deduce che il numero di archi sarà almeno $nk/2$. L'altra osservazione importante è che se non eliminiamo archi del taglio minimo, esso non può diminuire.

Consideriamo l'iterazione i-esima. Il numero di nodi n_i rimanenti nel multigrafo è pari a $n-i+1$ e se non abbiamo eliminato alcun nodo di K, la cardinalità k del taglio minimo è rimasta invariata. La probabilità di scegliere un arco in K sarà

$$\mathbb{P}(\text{scegliamo un arco} \in K) \leq \frac{k}{k \cdot n_i/2} = \frac{2}{n_i}.$$

La probabilità che l'algoritmo dia il risultato corretto è dunque

$$\begin{aligned}\mathbb{P}(C = K) &\geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n_i}\right) = \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \\ &= \prod_{i=1}^{n-2}\left(\frac{n-i+1-2}{n-i+1}\right) = \prod_{j=3}^{n}\left(\frac{j-2}{2}\right) = \frac{1}{\binom{n}{2}}\end{aligned}$$

che è limitato inferiormente da $\Omega(n^2)$. L'operazione di contrazione ha complessità $O(m)$ dove $m = |E|$. Se quindi operiamo $n^2/2$ iterazioni, il costo globale dell'algoritmo diventa $O(n^4)$ e la probabilità di errore è pari a

$$\left(1 - \frac{2}{n_i}\right)^{n^2/2} < \frac{1}{e}.$$

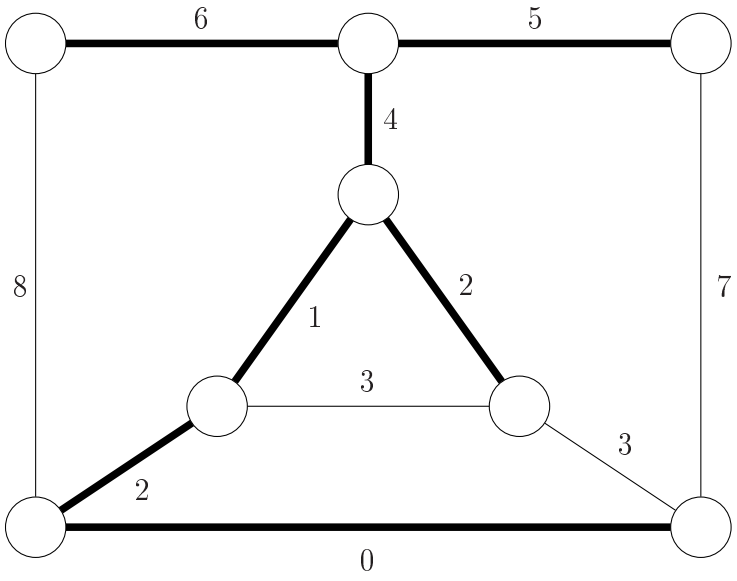
Con $n^2 \log n$ iterazioni, la probabilità di errore decresce all'aumentare di n.

Capitolo IV

Introduction to Trees

Let $G = (V, E)$ be an undirected graph with weights $W(i, j) \geq 0$, $\forall$\{i,j\} $\in E$; we denote by $n = |V|$ the number of vertices and $m = |E|$ is the number of edges.

If G does not contain cycles (there are no paths which connect each vertex v to itself), we can call G a *tree*.



IV.1 Spanning Tree Problem

Let's take into account a *cyclic* and *connected* graph $G = (V, E)$ with weights $W(i, j) \geq 0$, $\forall$\{i,j\} $\in E$. G is clearly not a tree, but we can eliminate some edges from G such as to render it acyclic. If we find a path which connects all the vertices of G without creating

cycles, we get a new sub-graph $G_T = (V, T)$ with $T \subseteq E$ which is surely a tree. We call it a *spanning tree*.

If G is *not connected*, we will end with a single spanning tree for each connected component of G: in that case we speak of *spanning forest*.

IV.1.1 Useful Definitions

Let's define some useful terms which will be used often in the report:

- F is a generic forest, defined as a set of trees.
- $F(x, y)$ is the unique path between x and y, if it exists in F (if they belong to the same connected component).
- $W_F(x, y)$ is the weight of the heaviest edge belonging to the path $F(x, y)$; if $F(x, y)$ does not exist, its value is conventionally ∞.
- $(x, y) \in E$ is called *F-heavy* if $W(x, y) > W_F(x, y)$.
- $(x, y) \in E$ is called *F-light* if $W(x, y) \leq W_F(x, y)$.

IV.1.2 Properties of Spanning Trees

Spanning trees have two very interesting properties, which are at the base of the algorithms computing the so-called *minimum spanning tree* (or *MST*).

Cycles: let C be a cycle in the starting graph G. We can be certain that the heaviest edge in C will not belong to the minimum weight spanning forest.

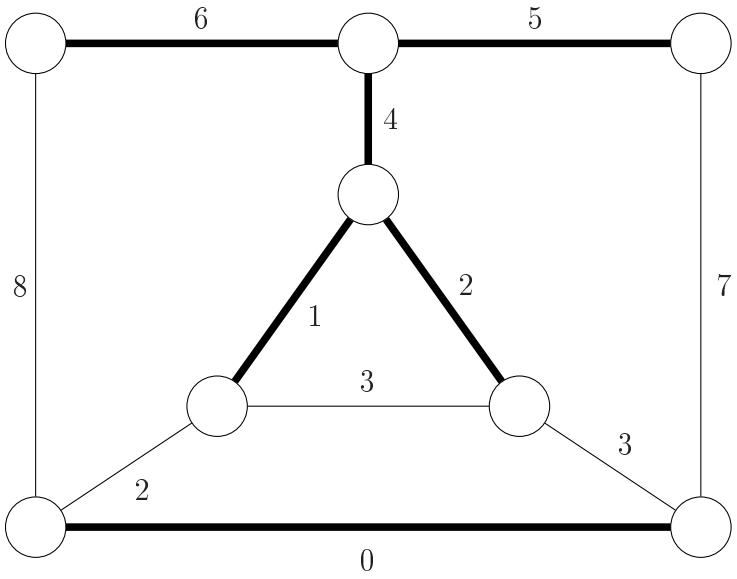
Cuts: let $X \subseteq V, X \neq \varnothing$ be a cut of the starting graph G. We can say that the lightest edge with a vertex in $\{X\}$ and the other vertex in $\{V \setminus X\}$ surely belongs to the minimum weight spanning forest.

IV.2 Algorithms for Minimum Weight Spanning Tree (MST)

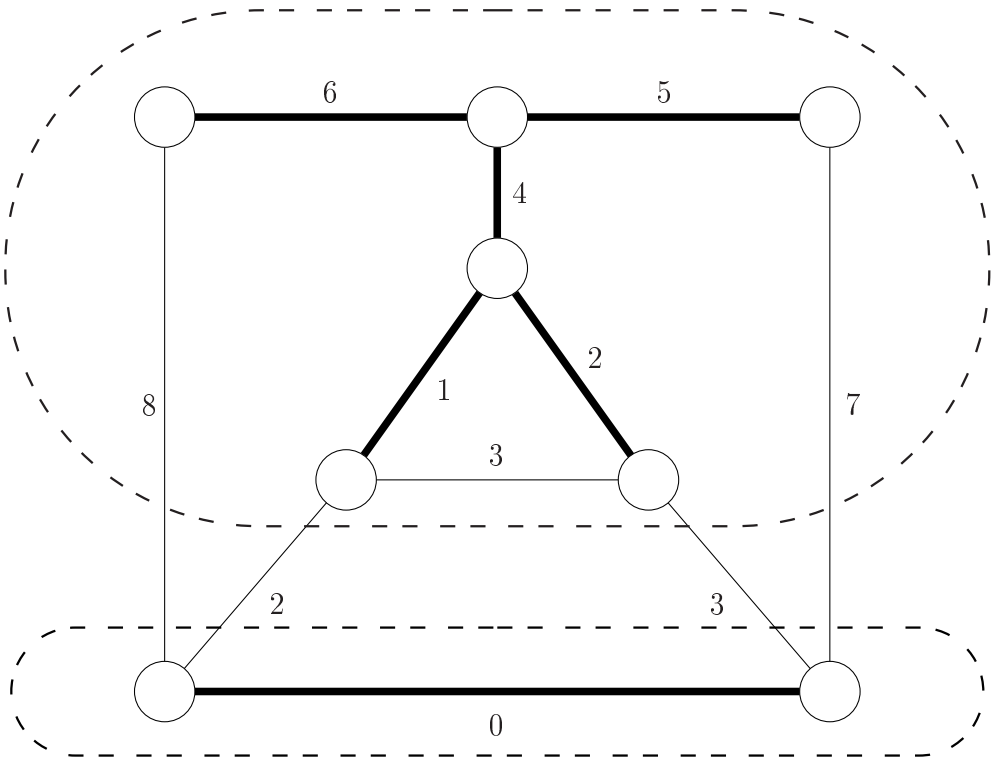
We will now present some of the various algorithms which have been developed to solve the minimum weight spanning forest problem. They will be described in chronological order.

IV.2.1 Boruvka Algorithm (1926)

This is probably the simplest but smartest algorithms of all, and its application is easy and straightforward; it takes advantage of the *cuts* property described above. For each vertex of the initial graph G we consider the cut which separates the given vertex v from the rest of the graph, and we put in the solution the lightest edge of the cut.

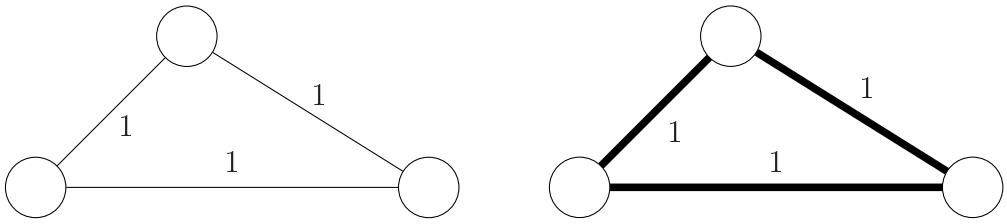


The solution is a partially connected graph. Then we collapse the connected vertices into a *macro-vertex*, and consider the cut which separates the macro-vertex from the rest of the graph: from this point on, we iterate the procedure until we get only one connected, acyclic component.



If we assume that the weights $W(x, y)$ with $x, y \in G$ are all different from each other, the solution is always correct (it finds the unique minimum spanning tree or forest) because

the algorithm takes advantage of the *cuts* property. If the assumption fails, the choosing of the minimum weight edge could become a problem if the given cut touches two edges with the same weight.



The solution used in the majority of implementations is the *lexico-graphical* order. We consider the vertices of the edges, and if $W(1,2) = W(2,3)$, we compare the number of the vertices: in this case $12 < 23$, so we choose W(1,2) as the minimum weight.

Algoritmo IV.1 BORUVKA(G, w, T)

1: $T \leftarrow \varnothing$
2: **while** $|T| < n - 1$ **do**
3: $F \leftarrow$ forest composed of minimum weight edges going out of each vertex
4: $G \leftarrow \{G/F\}$
5: $T \leftarrow T \cup F$

It is interesting to note that the algorithm is *parallel* and *distributed*: during each iteration we have to implement the same operation on all the vertices of the temporary solution, and we can easily do this in parallel, distributing the computation among many different executors.

IV.2.1.1 Complexity Analysis

For each iteration we have to:

- Find the forest composed by the lightest edge coming out of each vertex, which can be done in $O(m)$.
- Collapse the connected components in the temporary result, which can be done in $O(m)$.
- Unite the temporary result with the preceding result, which can be done in $O(1)$.

So, the single iteration is $O(m)$. But how many iterations we have to do? In the worst case, each vertex selects an edge which has been already selected by its neighbor. In this

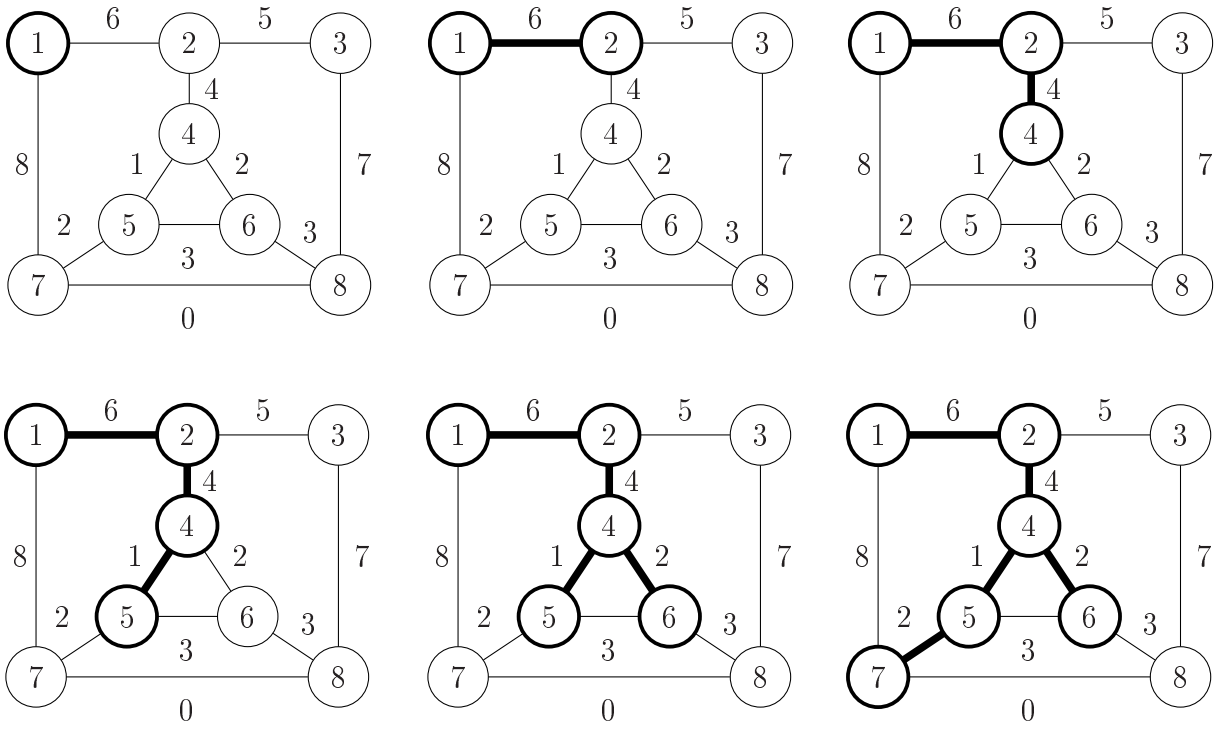
case we have $n/2$ connected components after the first iteration, $n/4$ after the second, $n/8$ after the third, and so on: we have to make $\lg n$ iterations.

The total complexity is then $O(m \lg n)$.

Note: an important characteristic of this algorithm is that we quickly eliminate a great number of heavy edges which will not belong to the MST.

IV.2.2 Prim Algorithm (1957)

We said that Borůvka's is a parallel algorithm: the Prim algorithm is just a sequential version of the Borůvka algorithm. The procedure is the same, but we work on a given sequence of vertices rather than on all of them together.



The correctness of the algorithm is granted by the *cuts* property, as for Borůvka (with the same tweak for edges having equal weights). The implementation is more detailed and straightforward, a bit similar to the Dijkstra procedure.

We need to define some new elements:

- r: root of the tree.

- $d[i]$: vertex label indicating the lightest edge connecting the vertex i to the partial solution.

- $p[i]$: vertex label indicating the *predecessor* or *father* of the vertex i.

Algoritmo IV.2 PRIM(G, w, T, r)

```
1: for all u ∈ V do
2:     d[u] ← ∞
3:     p[u] ← ∅
4: d[r] ← 0
5: p[r] ← r
6: Q ← V
7: while Q ≠ ∅ do
8:     u ← vertex with minimum d[i], i ∈ Q
9:     Q ← Q \ {u}
10:    for all v ∈ vertices next to u do
11:        if v ∈ Q and W(u, v) < d[v] then
12:            d[v] ← W(u, v)
13:            p[v] ← u
```

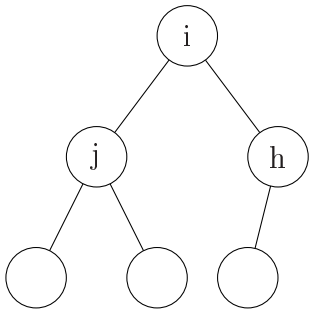
IV.2.2.1 Complexity Analysis

The crucial element is the implementation of Q, the set of vertices still to be included in the solution. On this set we have to iteratively apply two different operations: the extraction of the minimum $d[i]$ vertex, and the update of all the $d[i]$ labels.

We consider two different implementations of Q:

Q	List	Binary Heap
Extraction of minimum	$O(n)$	$O(\lg n)$
Update of labels	$O(1)$	$O(\lg n)$

The *binary heap* is often implemented as a binary balanced tree:



In the worst case analysis, eliminating the root (extracting the minimum label vertex) implies making $\lg n$ checks to determine the new minimum label. Updating the label of a vertex (*updating*, in this case, means *diminishing*) can make the vertex shift up some levels: in the worst case, it can shift from leaf to root, thus having to make $\lg n$ shifts.

To determine which implementation is better we have to consider how many times we make each operation.

- Extraction of minimum label vertex: we have to do it for each vertex, so n times.
- Update of labels: we have to do it m times in the worst case.

The total complexity then is:

List: $O(nn + m) = O(n^2)$

Binary heap: $O(n \lg n + m \lg n) = O(m \lg n)$

There is not an absolutely best implementation: we can note that if $m \simeq n$ (sparse graph) the binary heap is better, but if $m > n$ (dense graph) the list is better.

IV.2.3 Kruskal Algorithm (1956)

This algorithm takes advantage of the *cycle* property. It sorts the edges by growing weight, adding them one by one to the solution *only if they do not form cycles*. This last condition is the critical point of the complexity analysis.

The basic idea for the implementation is making many singleton sets each containing a vertex, and when we add an edge to the solution we collapse the two vertices into a single set. If, later in the sequence of edges, we find an edge which connects two collapsed vertices, we ignore it.

Algoritmo IV.3 KRUSKAL(G, w, T)

```
1: T ← ∅
2: for all v ∈ V do
3:    MakeSet(v)
4: Sort(E, w)
5: for all (u, v) ∈ E do
6:    if FindSet(u) ≠ FindSet(v) then
7:       T ← {u, v} ∪ T
8:       Union(u, v)
```

Apart from the SORT(E, w), which sorts the edges in growing order and can be implemented in many different ways (*QuickSort*, *BubbleSort*, *MergeSort* and so on), we have to define three new functions:

MAKESET(v)**:** creates a set given the vertex v and the data structure to hold the set (for example a list).

FINDSET(v)**:** returns the representative element of the set to which v belongs.

UNION(u, v)**:** unites the two sets to which u and v belong.

IV.2.3.1 Complexity Analysis

The critical point is the implementation of the three functions described above, and the data structure used to implement the sets.

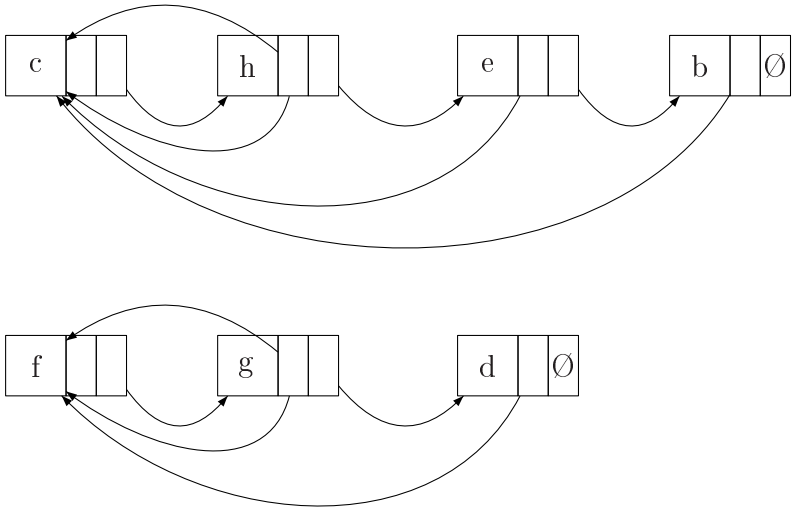
Let's consider an example of two disjoint sets:

$\{c, h, e, b\}$
$\{f, g, d\}$

We want to do the following operation:

if (FindSet(h) $\neq$ FindSet(b)) **then** Union(b, h)

Let's start by considering the most straightforward implementation of a set: the *list*. In this case, we can think of an implementation with pointers like this:



Each element has a pointer to the representative of the set, which we define as the first element of the set.

Let's analyze the cost of each operation with this implementation:

Sort: let's choose the *MergeSort* implementation $\Rightarrow O(m \lg m)$ with $m \leq n^2 \Rightarrow O(m \lg n^2) = O(2m \lg n) = O(m \lg n)$.

MakeSet: create a new list of one element $\Rightarrow O(1)$.

FindSet: each element has a pointer to the representative of the set $\Rightarrow O(1)$.

Union: we have to modify the pointers to the representative for each element in one of the two sets $\Rightarrow O(n)$.

Now let's consider the number of repetitions of each operation:

Operation	**Cost**	**Number of repetitions**	**Notes**
Sort	$O(m \lg n)$	1	None
MakeSet	$O(1)$	n	We create a singleton set for each vertex at the beginning of the algorithm.
FindSet	$O(1)$	$2m$	We call it twice for each iteration.
Union	$O(n)$	$n-1$	Each time we add an edge, we must unite the two sets.

The total complexity then is $O(m \lg n + n + 2m + (n-1)n) = O(m \lg n + n^2)$. We observe that the MakeSet and the FindSet cost less than the Sort, and that it is the Union which creates the most relevant problems. Can we do better than this? There are some upgrades which can lower the complexity of the algorithm, both on the Union alone and on the data structure. Let's consider them one by one.

IV.2.3.2 Weighted Union

In the preceding analysis we considered the worst case for the Union, in which both sets contained $n/2$ elements. But we know that worst case analysis can be easily beaten by *middle case* analysis. More often than not, we will have to unite two sets of different cardinality. If we suppose to have two sets S_1 and S_2, where $|S_1| = n_1$ and $|S_2| = n_2$ and we suppose that $n_1 > n_2$, it is surely quicker to update all the n_2 pointers in S_2. This selection of the shorter set is called *weighted union*. To analyze its cost we must use amortized complexity techniques, counting how many times we update the pointer to the representative element:

- 2 sets of 1 element each $\rightarrow$ 1 set of 2 elements, 1 update
- 2 sets of 2 elements each $\rightarrow$ 1 set of 4 elements, 2 updates
- $\vdots$
- 2 sets of $n/2$ elements each $\rightarrow$ 1 set of n elements, $n/2$ updates

If we sum all the costs iteration by iteration, we see that the cost of a single Union is $O(\lg n)$. As we can see from the table above, we have to do n Unions, so the total cost is $O(n \lg n)$, which is better than $O(n^2)$.

The complexity of the algorithm is then $O(m \lg n + n + 2m + n \lg n)$. We see that the critical point is now the Sort.

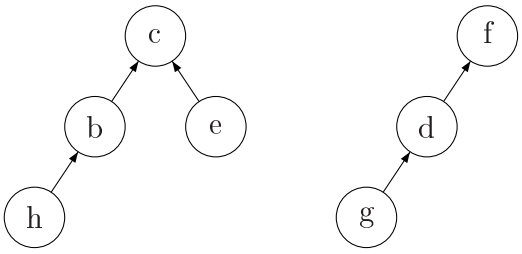
IV.2.3.3 Tree Structure

Now let's consider the case in which the edges are already sorted. Our complexity using the *weighted union* is then $O(n \lg n)$. Can we do even better than this? We can, but we must alter the data structure, moving from a list with pointers to the representative to a list with pointers to the predecessor (creating a tree).

We can easily see that this modification raises the cost of the FindSet from $O(1)$ to $O(h)$, where h is the height of the tree: if we do FindSet(v) and v is a leaf, we must climb all the tree up to the root, which is the representative. We will demonstrate that, with some careful adjustments, the gain on the complexity of the Union is higher than the loss on the FindSet.

A first observation is that if the tree is low and flat, i.e., it has few levels each containing many vertices, $O(h)$ is low because h is low. Our first aim will then be to maintain the tree as low as possible.

Let's consider the same example as before, with the two disjoint sets. This time, however, the data structure is a tree with pointers to the predecessors, and the root is the representative of the set.



Let's see a first simple implementation of the FindSet, using the formerly defined notation where $p[x]$ is the predecessor of the vertex x.

Algoritmo IV.4 FindSet(x)

```
1: if x ≠ p[x] then
2:     return FindSet(p[x])
3: else
4:     return x
```

For example, if we do FindSet(b), we climb up to h, then to c, and then we return c because it is the root.

The Union is very simple: if we do Union(c, f) we take the pointer from f to itself and we make it point to c. It is easy to note that, because we want to keep the height of a tree as low as possible, it's wiser to attach low trees to higher ones, so that the height is unaffected by the Union. With this adjustment, the complexity of the Union is $O(1)$.

Now let's consider *dynamic* trees: once every given number of iterations we adjust the structure of the tree, modifying some pointers to keep it as low as possible. In this case,

keeping track of the exact height of the tree can become costly. To solve this problem we define the *rank* of a tree: it is an excess estimation of the height of a tree. We raise the rank by 1 when there is equality between the ranks of two trees. To clarify this, here are the implementations of the three functions using the rank.

Algoritmo IV.5 MAKESET(x)

```
1: p[x] ← x
2: rank[x] ← 0
```

Algoritmo IV.6 FINDSET(x)

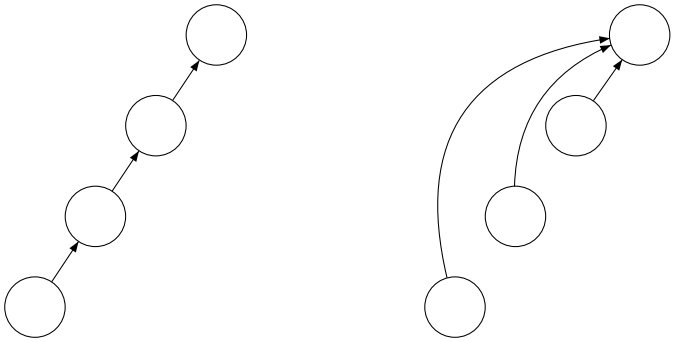
```
1: if x ≠ p[x] then
2:     p[x] ← FINDSET(p[x])
3: return p[x]
```

Algoritmo IV.7 UNION(x, y)

```
1: if rank[x] > rank[y] then
2:     p[y] ← x
3: else
4:     p[x] ← y
5:     if rank[x] = rank[y] then
6:         rank[y] ← rank[y] + 1
```

In the worst case, the so-called *Union-by-Rank* unites simple lists, so its cost is always $O(\lg n)$.

We have seen how the rank is raised; can it be lowered? Yes, if we introduce the concept of *path compression*. If we make a FINDSET(b) on the example below, we climb up the tree until we get to the root c. When we climb up the predecessors, we can bring with us the pointers and make them point directly to the root, obtaining a hybrid structure between the tree presented here and the list with pointers to the representative presented before.



If we add the *path compression*, the cost of all the FindSETs goes from $O(mh)$ to $O(m\alpha(n))$, where $\alpha(n)$ is the *Ackermann Inverse*. For reasonable values of n, we see that $0 \leq \alpha(n) \leq 4$. To demonstrate why, let's first consider the *Ackermann Function*:

(IV.1)
$$A_k(j) = \begin{cases} j+1 & \text{if} k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if} k \geq 1 \end{cases}$$

where $A_{k-1}^{(n)}(j)$ is the application of $A_{k-1}(j)$ n times.
To understand better, let's see a numerical example:

$$\begin{aligned} A_{k-1}^{(0)}(j) &= j \\ A_{k-1}^{(1)}(j) &= A_{k-1}(A_{k-1}^{(j-1)}(j)) \\ A_1(j) &= 2j+1 \\ A_2(j) &= 2^{j+1}(j+1) - 1 \\ A_3(1) &= A_2^{(2)}(1) = ... = A_7^{(2)}(j) = 2047 \\ A_4(1) &= A_2^{(2048)}(2047) \gg 10^{80} \end{aligned}$$

Now let's define the *Ackermann Inverse*:

(IV.2)
$$\alpha(n) = min\{k : A_k(1) \geq n\}$$

then if $n \leq 2047 \Rightarrow \alpha(n) = 3$ and if $n \leq 10^{80} \Rightarrow \alpha(n) = 4$

If we consider that 10^{80} should be the number of vertices in our graph and that 10^{80} is greater than the number of molecules in the universe, we can be fairly certain that $\alpha(n)$ will always be no more than 4. The growth is not exactly linear, but is close to linear.

IV.2.4 RandomMST Algorithm

For Kruskal we found a nearly-linear implementation, but we saw it is really difficult to go lower than that. The *Random MST Algorithm* takes advantage of the randomized complexity to operate at a linear cost. We saw that the Borůvka algorithm behaves well in the first iterations: it quickly eliminates many unnecessary edges. Going ahead, though, its efficiency lowers. We can take advantage of this behavior by making some iterations of the Borůvka algorithm (namely 3) and then proceeding with another procedure on a partial solution whose dimension is much lower than the initial one.

To fully understand the workings of the algorithm it is necessary to remind some probability computation definitions:

Bernoullian Distribution It models the result of a coin toss.
(IV.3)

$$f(x) = \begin{cases} p & \text{if x = 0} \\ 1-p & \text{if x = 1} \\ 0 & \text{otherwise} \end{cases} \qquad with \qquad E[X] = p$$

Binomial Distribution It is the sum of n Bernoullian distributions.
(IV.4)

$$f(x) = \binom{n}{x} p^x (1-p)^{(n-x)} \qquad with \qquad E[X] = np$$

Geometrical Distribution It is the number of realizations of a Bernoullian distribution necessary to have at least one head.
(IV.5)

$$f(x) = \begin{cases} p(1-p)^{x-1} & \text{if x = 1,2,...} \\ 0 & \text{otherwise} \end{cases} \qquad with \qquad E[X] = 1/p$$

Negative Geometrical Distribution
It is unnecessary to describe it analytical form here; it is the number of tosses we must make to get n heads, and its expected value is $E[X] = n/p$.

Another concept to define is the *random sampling*: given a graph G, we call $G(p)$ a graph with all the vertices of G and with a random number of edges between those in G, each selected with probability p. Thus, if G has n vertices and m edges, $G(p)$ will have n vertices and $p \cdot m$ expected edges.

Now let's see the algorithm by phases:

Algoritmo IV.8 RANDOM MST(G, w, F)

1: Make 3 iterations of Borůvka algorithm → we have G_1 with $n/8$ vertices and C chosen edges.
2: *Random sampling* with $p = 1/2 \rightarrow G_2 = G_1(p)$ with $n/2$ expected edges.
3: RANDOM_MST(G_2, w, F_2)
4: Eliminate $F_2 - heavy$ edges from $G_1 \rightarrow G_3$
5: RANDOM_MST(G_3, w, F_3)
6: $F \leftarrow F_3 \cup C$

IV.2.4.1 Complexity Analysis

The critical point is the fourth phase of the algorithm: here we eliminate $F_2 - heavy$ edges from G_1, but because F_2 comes from a random sampling of G_2 and G_2 itself comes from a

random sampling of G_1, we could have lost some critical edges on the run. We know that in MST problems, *critical edges* are F-light edges. Considering the negative geometrical distribution defined above, if we can prove that the expected number of F-light edges in F is less than or equal to n/p, we can be sure that the algorithm will be linear.

Let's consider F = MST of $G(p)$, and let's take $e_1, e_2, \ldots e_m$ the edges sorted by growing weight. F, at the beginning of the algorithm, is empty. The edge e_i is chosen with probability p: if it belongs to F in this step (so it is F-light), we can be sure that it will belong to F also at the end of the algorithm (it will remain F-light until the end).

We divide the building of the solution on phases: the phase k begins after we add the $k-1$-th edge to F, and ends when we add the k-th. With this division, we know that we must wait x steps before adding a new edge to the solution, and to each step we have a probability of p of adding an edge to the solution. This is like tossing a coin many times and waiting for the first head to show up: it is a geometrical distribution with parameter p.

If the solution has s edges (which must be all obviously F-light), we can be sure that the above phases will be exactly s, with $s \leq n-1$. As we have said that each phase is a geometrical distribution of probability, we have the sum of s geometrical distributions, which is a negative geometrical distribution. Our expected value is s/p, but because $s \leq n-1$, we can consider that $s \leq n$ and say that $E[x] \leq n/p$.

Now that we proved that the expected number of F-light edges in the solution is less than or equal to n/p, let's consider the complexity of each phase of the algorithm, using the techniques for recursive complexity.

1. $O(n+m) \leftarrow$ 3 iterations of Borůvka's algorithm can be considered linear

2. $O(m) \leftarrow$ *random sampling*, we must simply visit all the edges

3. $T(n/8, m/2) \leftarrow$ recursive call with $n/8$ vertices because of the 3 iterations of Borůvka and with $m/2$ edges because of random sampling with $p = 1/2$

4. $O(n/8 + m) \leftarrow$ we still have $n/8$ vertices, but we are working on G_1 and not on $G_1(p)$ so we still have m edges; we keep $n/8p$ edges, so we keep $n/4$ edges because $p = 1/2$

5. $T(n/8, m/4) \leftarrow$ we have m$/4$ edges because $O(n) = O(m)$

6. $O(n+m)$

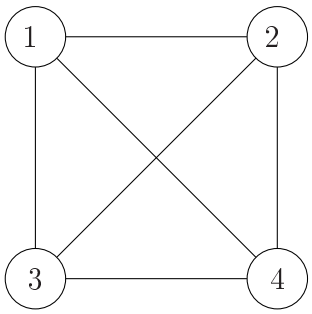
We can see that

(IV.6)
$$T(n,m) \leq T(n/8, m/2) + T(n/8, m/4) + c(n+m) \leq 2c(n+m)$$

As we can see, the total complexity is $O(n)$.

IV.3 Structure Enumeration

This problem is another common and interesting application of trees. Suppose we have a complete graph like this:



We want to *enumerate all the spanning trees of the graph.* To solve the problem, we need to find a code which can univocally represent each solution (in this case, each spanning tree). When we have that code, we can easily enumerate the solutions by *decoding* each code in the sequence of all the solutions.

IV.3.1 Prüfer Sequence

If we want to code all the spanning trees of a given graph, we can use the *Prüfer Sequence*. It is a code which can be applied to any labelled tree, and it creates a sequence of labels which univocally defines the tree.

The Prüfer sequence is a set of $(n-2)$ numbers $\in \{1 \dots n\}$ in which the numbers are *vertex labels* of our tree. To manage the correspondence between tree and sequence we need a CODE and a DECODE function.

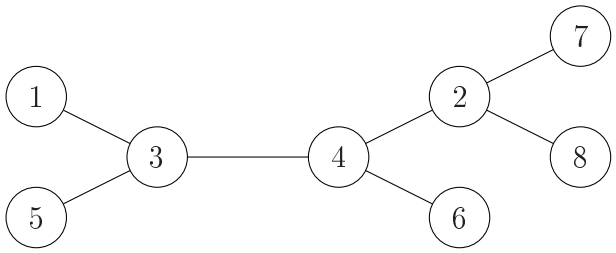
We use the notation where $p[v]$ is the *predecessor* of v and $head(S)$ is the head of the sequence.

Algoritmo IV.9 CODE(T)

```
1: if T is an edge then
2:     return ∅
3: else
4:     V ← leaf vertex of minimum label
5:     return p[v] ⊕ CODE(T \ {V})
```

Algoritmo IV.10 DECODE(S, N)

```
1: if |N| = 2 then
2:     return {v_1, v_2}
3: else
4:     v ← vertex in N but not in S with minimum label
5:     return {v, head(S)} ⊕ DECODE(S ⊖ head(S), N \ {v})
```



It can be proved that we can generate n^{n-2} different sequences given a graph with n vertices

IV.4 Algorithms for the enumeration of limited degree trees

First of all, we must define what we mean as *degree* of a vertex. Given a tree $T = (V, E)$ and a vertex $u \in V$, we define the *degree* of u as

$$d(u) = |\{v \in V : (u, v) \in E\}|$$

which is the number of edges going out of the vertex. Given an integer K, we define a *limited degree tree* $T_K = (V, E)$ a tree in which

$$\forall u \in V \qquad d(u) \leq K.$$

So we take into account only trees whose vertices have at most K edges going out.

We want to consider two different but linked problems:

Counting Problem Given $n = |V|$ and K integer, **count** all the different trees $T_K = (V, E)$.

Enumeration Problem Given $n = |V|$ and K integer, **enumerate** (*enumerate* means *count* and *describe the structure*) all the different trees $T_K = (V, E)$.

We can see that even if we consider a low K, the dimension of the problem quickly becomes huge: for $K = 4$, if we have 12 vertices we have 355 different trees; but if we have 20 vertices, we already have 366319 trees.

Osservazione IV.1. The problem of counting and enumerating trees of limited degree first came out along with the classification of the alkane molecules, in the beginning of the XIX century. Some scientists who worked at the problem were *Berzelius* (1830), *Jordan* (1869) and *Cayley* (1875). □

IV.4.1 N-tuple Code

As we saw that the number of trees to enumerate can quickly become huge, we need to find a *code* which can univocally represent a tree while keeping disk and memory occupation low. To solve this problem, the *N-tuple* code has been developed.

Osservazione IV.2. The *N-tuple* code uses the *lexico-graphical* ordering to ensure uniqueness of each coding. For ease of comprehension, we define the lexico-graphical ordering as follows: Given two sequences of integers

$$C = c_1 \dots c_l$$

and

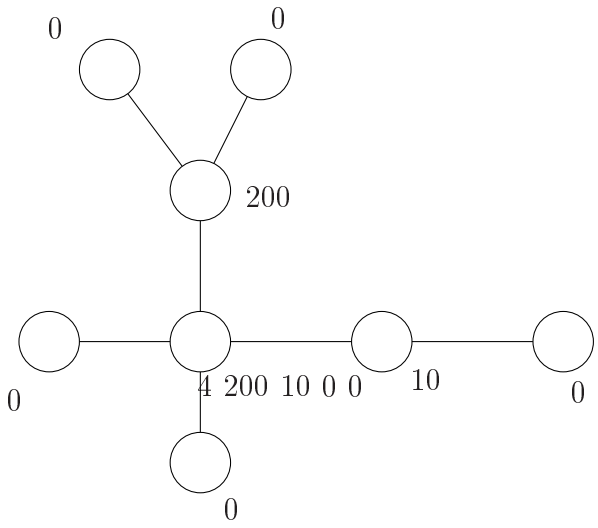
$$C' = c'_1 \dots c'_{l'}$$

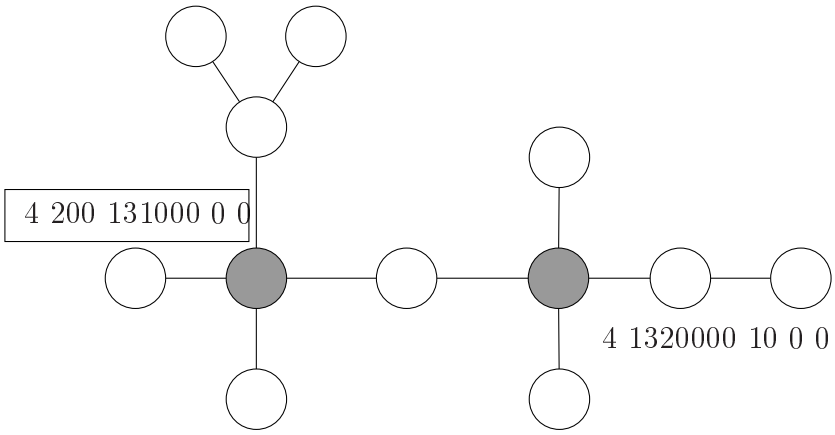
we say that $C > C'$ if $\exists j : 1 \leq j \leq min\{l, l'\}$ so that $c_i = c'_i$ for $i = 1, \dots, j-1$ and $c_j > c'_j$ or if $l > l'$ and $c_i = c'_i \forall i = 1, \dots, l'$

Esempio IV.1. In lexico-graphical order $956 > 1235$, and $956 > 95$. ■

We can define the *N-tuple* encoding procedure of a **rooted** tree with root r as follows:

- every leaf has code 0
- let g be the number of vertices $(v_1, \dots, v_g)$ adjacent to the root r
- delete r obtaining g sub-trees, of which we compute the *N-tuple* code
- compose the codes of the g sub-trees obtaining a sequence S which is maximum by lexico-graphical order
- the final code C of the tree is the composition of g and S





The encoding procedure of a **non-rooted** tree is as follows:

- let M be the set of vertices with maximum degree
- compute the code C_u for each $u \in M$
- the tree code C is $C = max\{C_u : u \in M\}$, where the maximum is in lexico-graphical ordering

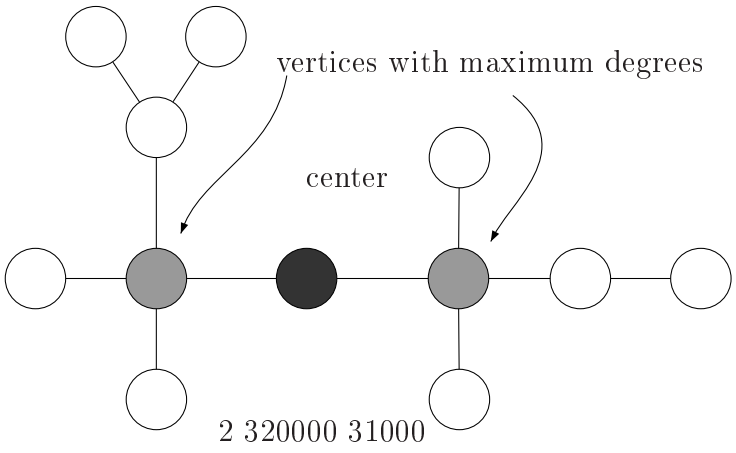
IV.4.2 Centered N-tuple Code

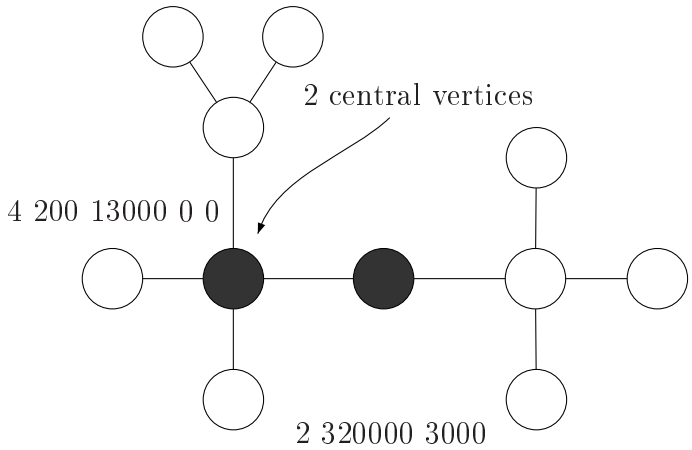
For **non-rooted** trees, computing the tree code C involves computing the code C_u of all the maximum degree vertices. To avoid worthless computation, we can obtain C by starting only from the *center* of the tree, which can be composed by 1 or 2 vertices at most.

Osservazione IV.3. We can find the *center* of a non-rooted tree T in a recursive way:

1. eliminate all the leaves and their edges from T, obtaining a smaller tree T'
2. if T' contains exactly 1 vertex or at most 2 adjacent vertices, it is the center
3. if not, repeat the process □

If we end with a two-vertices center, we compute the N-tuple code for each of them and we choose the lexico-graphical maximum.





Lemma IV.1. *The complexity of the computation is $O(n^2)$ for N-tuple code and $O(n \lg n)$ for the centered N-tuple code.*

Dimostrazione. To obtain a generic N-tuple code we must visit all the vertices of the tree once, and we must make a certain number of comparisons to determine the lexico-graphical maximum.

For the non centered N-tuple code we have:

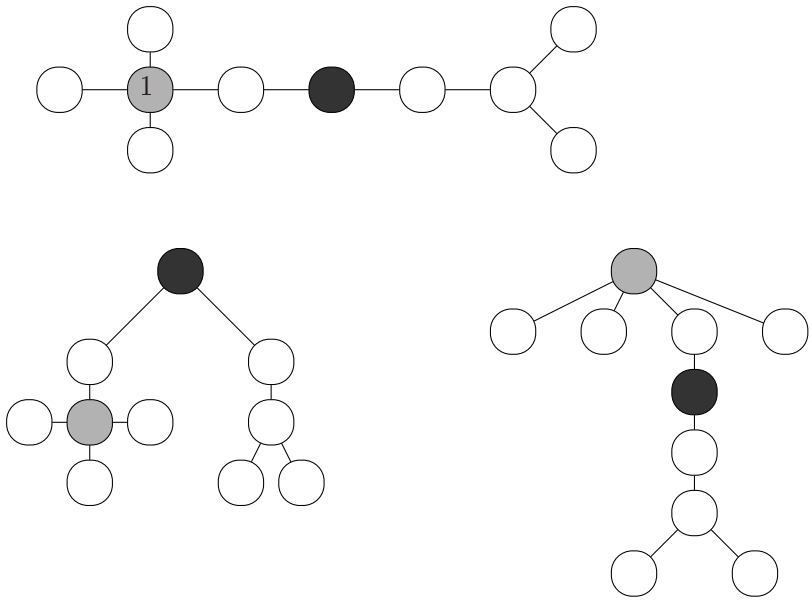
- visit of the tree $\rightarrow O(n)$
- the tree is not balanced $\rightarrow O(n)$ comparisons

So its complexity is $O(n) \times O(n) = O(n^2)$

For the centered N-tuple code we have:

- visit of the tree $\rightarrow O(n)$
- the tree is balanced $\rightarrow O(\lg n)$ comparisons □

So its complexity is $O(n) \times O(\lg n) = O(n \lg n)$



IV.4.3 One-to-one Enumeration

This is probably the most intuitive enumeration algorithm one can think of: it lists the trees with n vertices by adding one vertex in all possible ways to the tree with $n-1$ vertices.

More formally, given a tree $T = (V, E)$ with $n-1$ vertices, and defined

$$N^+ = \{u \in N : d(u) < K\},$$

we can say that by adding 1 vertex to T we obtain exactly $|N^+|$ trees with n vertices.

We must also define the notation which will be used from here on:

- T_o: origin tree of an enumeration
- T_d: tree generated from T_o

- *Tree Generation Function:*

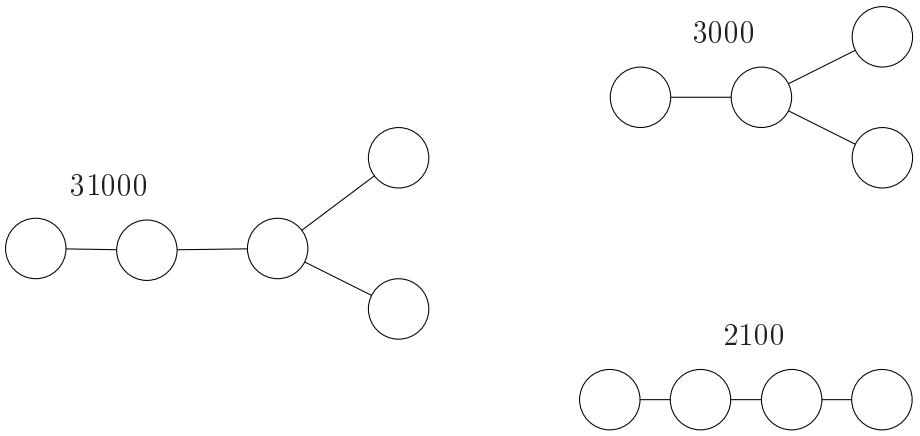
$$G_T : T_o \longmapsto T_d$$

- *Multi-sets:*

$$D(T_o) = \{T_d : G_T(T_o) = T_d\}$$

$$O(T_d) = \{T_o : G_T^{-1}(T_d) = T_o\}$$

This kind of enumeration can lead to non-unique tree coding:



To avoid this, we introduce *Reverse Search* (RS) and *Symmetry Detection* (SD).

IV.4.3.1 Reverse Search

We can avoid the enumeration of the same tree from different trees by applying the so-called *Enumeration Rule 1*:

Enumeration Rule 1: a tree $T_d = G_T(T_o)$ is valid if and only if T_o is the lexico-graphical maximum in $O(T_d)$.

IV.4.3.2 Symmetry Detection

This rule avoids generating identical trees by adding one vertex in different positions to the origin tree. First of all we must define *symmetric* trees. Let T_o be a tree with root r and let S_1 and S_2 be two sub-trees with roots respectively in v and w. If we define $l_r(v)$ the distance in number of edges between r and v, we can say that S_1 and S_2 are symmetric if and only if

- their two codes are identical
- $l_r(v) = l_r(w)$

Given this, we can avoid symmetry by applying *Enumeration Rule 2*:

Enumeration Rule 2: given S_i symmetric sub-trees of T_o with $i = 1, \ldots, q$ and $2 \leq q \leq 4$, the trees generated by T_o are only those obtained by adding nodes to S_q.

IV.4.3.3 CTree

If we want to implement efficiently the two enumeration rules, we need to define a new data structure called *Code Tree*, or simply *CTree*. It is a tree rooted in the vertex where we are computing the code, and it is composed of levels so that at level 0 we have the whole code of the tree, and at level i we have the codes of the trees at distance i from r.

Knowing the CTree of T_o, it is easy to generate the code of T_d because we only have to add a leaf with code 0 to the CTree and climb up the levels modifying the codes of the relative sub-tree. The levels of the CTree are sorted by lexico-graphical order, so it's easy to exchange positions between brothers who have changed their codes.

Let's have a brief look at the procedures that enumerate all the trees using one-to-one enumeration:

Generation of $D(T_o)$

- generate the CTree of T_o
- execute a BFS visit of the CTree eliminating symmetries (brothers with the same code)
- at each step of the BFS, if the vertex u is valid and $d(u) < K$, generate T_d and store it in $D(T_o)$ only if the enumeration rule 1 is satisfied, then add to the visit queue the children of u
- return $D(T_o)$

One-to-one Enumeration Algorithm

- initialize the set of generated trees with $\{10\}$, the set of trees with 2 vertices
- at each step $i = \{3, \ldots, n\}$
  - extract a tree T_o with $i - 1$ vertices
  - generate $D(T_o)$ with the procedure above
  - add $D(T_o)$ to the set of trees with i vertices and update the counter
- stop when we extract all the trees with $i - 1$ vertices

IV.4.4 Constructive Enumeration

Instead of generating *every* tree with n vertices by adding vertices to smaller trees, we can get a n vertices tree by adding **one vertex** to a set of sub-trees whose total number of vertices is $n-1$. The number and cardinality of these sub-trees is easily obtained using *Jordan's Theorem*, which describes how to meld rooted trees to obtain non-rooted trees.

Jordan's Theorem (1869) Let T be a tree with n vertices:

- for $n = 2k+1$ odd, exists a single vertex called **centroid** so that all the (2 or more) incident sub-trees have at most k vertices
- for $n = 2k$ fair, can exist:
  - a single vertex called **centroid** so that all the (3 or more) incident sub-trees have at most k vertices
  - a single edges called **bi-centroid** so that the two incident sub-trees have exactly k vertices

Given this, we can divide the procedure in two separate steps:

- find 3 or 4 numbers whose sum is $n-1$;
- combine in all possible ways the sub-trees with cardinality equal to the numbers found until we get a tree with exactly n vertices (by adding the centroid).

By comparing the complexities of four different algorithms which use four different combinations of the characteristics presented here, we can sum up some conclusions about performance. We analyze the following four algorithms:

Name	Kind of enumeration	Avoidance of same tree generation	Code used
TNKMS	one-to-one	list of generated trees	N-tuple
HPVK	one-to-one	RD + SD	N-tuple
AHM	one-to-one	RD + SD	CN-tuple
KP	constructive	not necessary	CN-tuple

We can draw the following conclusions:

- TNKMS is the oldest and worse, it can't enumerate trees with more than 20 vertices (the processing time becomes excessive) and it's difficult to analyze its true complexity as it is an old Fortran implementation
- HPVK and AHM are comparable, but they must enumerate all the n non-rooted trees because they use one-to-one enumeration
- KP is by far the best, because it can avoid to ensure that the newly generated trees hasn't been generated before (uniqueness ensured by Jordan's Theorem), and because it works on rooted trees.