Algoritmi e Strutture Dati

Elisa Solinas
Corso del prof. Ugo De' Liguoro

A.A. 2016/2017
Updated on 6 giugno 2017

Indice

Capitolo 1

Notazione asintotica

1.1 Notazione Θ
Limite asintoticamente stretto

$$f(n) \in \Theta(g(n)) \qquad \leftrightarrow \qquad \exists c_1, c_2, n_0 : \quad \forall n \geq n_0 \\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \tag{1.1}$$

Informalmente: Esistono due costanti c_1, c_2 tali che $f(n)$ possa essere rinchiusa tra $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$

1.2 Notazione O-grande
Limite asintotico superiore

$$f(n) \in O(g(n)) \qquad \leftrightarrow \qquad \exists c, n_0 : \quad \forall n \geq n_0 \\ 0 \leq f(n) \leq c \cdot g(n) \tag{1.2}$$

Informalmente: $g(n)$ è un limite superiore a $f(n)$ a meno di una costante.

1.3 Notazione Ω
Limite asintotico inferiore

$$f(n) \in O(g(n)) \qquad \leftrightarrow \qquad \exists c, n_0 : \quad \forall n \geq n_0 \\ 0 \leq c \cdot g(n) \leq f(n) \tag{1.3}$$

Informalmente: $g(n)$ è un limite inferiore a $f(n)$ a meno di una costante.

1.4 Notazione o-piccolo
Limite asintotico superiore non stretto

$$f(n) \in O(g(n)) \qquad \leftrightarrow \qquad \forall c \exists n_0 : \quad \forall n \geq n_0 \\ 0 \leq f(n) \leq c \cdot g(n) \tag{1.4}$$

Informalmente: La funzione $f(n)$ diventa insignificante rispetto a $g(n)$ quando n tende a ∞:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \tag{1.5}$$

Capitolo 2

Relazioni di ricorrenza

2.1 Il metodo dell'esperto

2.1.1 Relazioni lineari a partizione costante

$$T(n) = \begin{cases} d & n \leq m \leq h \\ \sum_{1 \leq i \leq h} a_i \cdot T(n-1) + c \cdot n^\beta & n > m \end{cases} \tag{2.1}$$

Consideriamo $c > 0, \beta \geq 0, a = \sum_{1 \leq i \leq h} a_i$:

$$\begin{cases} T(n) \in O(n^{\beta+1}) & a = 1 \\ T(n) \in O(a^n \cdot n^\beta) & a \geq 2 \end{cases} \tag{2.2}$$

2.1.2 Relazioni lineari a partizione bilanciata

$$T(n) = \begin{cases} d & n = 1 \\ a \cdot T(\frac{n}{b}) + c \cdot n^\beta & n > 1 \end{cases} \tag{2.3}$$

Consideriamo $a \geq 1, b \geq 2, c > 0, d, \beta \geq 0, \alpha = \frac{\log a}{\log b}$:

$$\begin{cases} T(n) \in O(n^\alpha) & \alpha > \beta \\ T(n) \in O(n^\alpha \log n) & \alpha = \beta \\ T(n) \in O(n^\beta) & \alpha < \beta \end{cases} \tag{2.4}$$

2.2 Svolgimento delle ricorrenze

2.2.1 Usare le sommatorie

$$\begin{gathered} \sum_{k=1}^{n} k = \frac{1}{2} \cdot n \cdot (n+1) \\ \sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \\ \sum_{k=1}^{n} \frac{1}{k} = \log_n n + O(1) \end{gathered} \tag{2.5}$$

Capitolo 3

Algoritmi di Ordinamento

3.1 Insertion Sort

3.2 Selection Sort

3.3 Quick Sort

3.4 Merge Sort

3.5 Counting Sort

Capitolo 4

Strutture Dati

4.1 Array statici

4.2 Array dinamici

4.3 Liste

4.4 Hashing Table

Capitolo 5

Code di priorità

Capitolo 6

Alberi

6.1 Alberi radicati

6.2 Alberi binari di ricerca

6.3 Alberi Rosso-Neri

Capitolo 7

Union-Find

Alcune applicazioni, come la rappresentazione dei grafi, richiedono di raggruppare n elementi distinti in una collezione di insiemi disgiunti.
Queste applicazioni richiedono anche l'esecuzione di due operazioni particolari:

- **Union**: unire due insiemi.
- **Find**: trovare l'unico insieme che contiene un determinato elemento.

7.1 Caratteristiche di una struttura Union-Find

Una struttura dati per insiemi disgiunti mantiene una collezione

$$S = \{S_1, ..., S_n\} \tag{7.1}$$

Ciascun insieme S_i viene identificato attraverso uno dei suoi elementi, detto **rappresentante** di S_i.
La scelta del rappresentante è effettuata in maniera differente a seconda dell'implementazione: in alcune tale scelta non è importante, mentre in altre il rappresentante di un insieme ha delle caratteristiche particolari (per esempio, è il minimo elemento). L'importante è che, se richiediamo due volte il rappresentante di un dato insieme (che non viene modificato fra le due richieste), la risposta dev'essere la stessa.

Le operazioni fondamentali di una Union-Find sono le seguenti:

- `Make-Set (x)`: crea un nuovo insieme il cui unico elemento (e rappresentante) è x.
- `Union (x, y)`: unisce gli insiemi dinamici che contengono $x \in S_x$ e $y \in S_y$ in un nuovo insieme che è l'unione dei due insiemi.
  In alcune applicazioni, il rappresentante dell'insieme risultante è un elemento qualsiasi di $S_x \cup S_y$, mentre in altre viene scelto specificamente il rappresentante di S_x o S_y.
  Spesso, nella pratica, gli elementi di uno degli insiemi vengono assorbiti dall'altro insieme.
- `Find-Set (x)`: restituisce un puntatore al rappresentante dell'insieme che contiene x.

7.2 Implementazione della Union-Find mediante foresta di alberi radicati

Rappresentiamo gli insiemi con **alberi radicati**, dove ogni nodo contiene un elemento e ogni albero rappresenta un insieme.
In una **foresta di alberi disgiunti**, ogni elemento punta soltanto a suo padre. Il nodo radice di ogni

albero contiene il rappresentante ed è padre di se stesso.

Le tre operazioni delle Union-Find vengno realizzate come segue:

- `MAKE-SET`: crea semplicemente un albero con un solo nodo.
- `FIND-SET`: segue i puntatori ai padri finché non trova la radice dell'albero. I nodi visitati in questo cammino semplice verso la radice costituiscono il **cammino di ricerca**.
- `UNION`: fa sì che la radice di un albero punti alla radice dell'altro albero.

7.2.1 Euristiche per migliorare il tempo di esecuzione

Unione per rango: l'idea consiste nel fare in modo che la radice dell'albero con meno nodi punti alla radice dell'albero con più nodi. Anziché tenere esplicitamente traccia della dimensione del sottoalbero radicato in ciascun nodo, per ogni nodo manteniamo un `rank` che è un limite superiore per l'altezza del nodo. Sfruttando questa euristica, nell'operazione di `UNION`, la radice con il rango più piccolo viene fatta puntare alla radice con il rango più grande.

Compressione del cammino: questa euristica viene utilizzata durante le operazioni `FIND-SET` per fare in modo che ciascun nodo nel cammino di ricerca punti direttamente alla radice. La compressione del cammino non cambia i ranghi.

7.2.2 Algoritmi

`MAKE-SET (x)`

Quando l'operazione `MAKE-SET` crea un nuovo insieme con un solo elemento, il rango iniziale dell'unico nodo è 0.

```
1  MAKE_SET (x)
       x.p = x
3      x.rank = 0
```

`UNION (x, y)`

Quando applichiamo l'operazione `UNION` a due alberi, si presentano due casi, a seconda che le radici abbiano o meno lo stesso rango.
Se le radici hanno lo stesso rango, trasformiamo la radice di rango più alto in padre della radice di rango più basso, ma i ranghi restano inalterati.
Se, invece, le radici hanno lo stesso rango, trasformiamo arbitrariamente una delle radici in padre e ne incrementiamo il rango.

```
1  UNION (x, y)
       LINK(FIND_SET(x), FIND_SET(y))
3  LINK (x,y)
       if (x.rank > y.rank)
5          y.p = x
       else
7          (x.p = y)
           if (x.rank == y.rank)
9              y.rank = y.rank + 1
```

`FIND-SET (x)`

L'operazione `FIND-SET` segue i puntatori ai padri finché non trova la radice dell'albero.
Ogni operazione `FIND-SET` lascia i ranghi inalterati.

```
FIND-SET (x)
    if (x ≠ x.p)
        x.p = FIND-SET(x.p)
    else
        return x.p
```

Questa procedura è un metodo a **doppio passaggio**: durante il primo passaggio risale il cammino di ricerca per trovare la radice; durante il secondo passaggio, discende il cammino di ricerca per aggiornare i nodi in modo che puntino direttamente alla radice.

7.3 Rappresentare grafi con le Union-Find

Una delle tante applicazioni delle Union-Find consiste nel determinare le componenti connesse di un grafo non orientato.

Le procedure fondamentali che permettono tali applicazioni sono le seguenti:

- `Connected-Components (G)`: usa le operazioni fondamentali per calcolare le componenti connesse di un grafo.
  Sostanzialmente, la procedura crea un nuovo insieme per ogni vertice di G. Successivamente, per ogni arco, unisce gli insiemi che contengono i due archi (se non si trovano già nello stesso insieme).

```
CONNECTED-COMPONENTS(G)
    forall (v ∈ G.V)
        MAKE-SET (v)
    forall ((u,v) ∈ G.E)
        if (FIND-SET(u) ≠ FIND-SET(v))
            UNION (u,v)
```

- `Same-Components (u,v)`: una volta che `Connected-Components (G)` ha pre-elaborato il grafo, questa procedura determina se due vertici sono nella stessa componente connessa.

```
SAME-COMPONENTS(u,v)
    if (FIND-SET(u) == FIND-SET(v))
        return true
    else
        return false
```

Capitolo 8

Grafi

8.1 Definizioni

Grafo orientato: $G(V, E)$

- V (**insieme dei vertici**): insieme finito di elementi.
- E (**insieme degli archi**): relazione binaria su elementi di V, indicati con (u, v)

Grafo non orientato: $G(V, E)$
L'insieme degli archi è composto da coppie non orientate, ovvero ogni arco è un insieme $\{u, v\}$ e non una relazione.

Se (u, v) è un arco in un grafo orientato $G = (V, E)$ diciamo che (u, v) **esce** dal vertice u ed **entra** nel vertice v.
Se (u, v) è un arco in un grafo non orientato $G = (V, E)$ diciamo che (u, v) è **incidente** nei vertici u e v.
Se (u, v) è un arco di un grafo $G = (V, E)$ diciamo che il vertice v è **adiacente** al vertice u. Se il grafo non è orientato, la relazione adiacenza è simmetrica.

Cappi: archi che entrano ed escono nello stesso vertice, del tipo (u, u). Non sono ammessi nei grafi orientati.

Cammino di lunghezza k da u a v: è una sequenza di vertici $< v_0, ..., v_k >$ tali che $v_0 = u$ e $v_k = v$

$$\forall i \in [1...k] \qquad (v_{i-1}, v_i) \in E \tag{8.1}$$

Ciò significa che per ogni v_{i-1} del cammino esiste un arco che lo collega a v_i.
k è la **lunghezza del cammino** ed è pari al numero di archi nel cammino.
Se esiste un cammino da u a v, diciamo che v è **raggiungibile** da u.
$\delta(u, v)$ dove u è un vertice e v è un altro vertice, è detto **cammino minimo** da u a v ed è così definito:

$$\delta(u, v) = \begin{cases} n & \text{lunghezza minima di un cammino da u a v se esiste} \\ \infty & \text{altrimenti} \end{cases} \tag{8.2}$$

Ciclo: in un grafo orientato un cammino $< v_0, ..., v_k >$ forma un ciclo se $v_0 = v_k$ e il cammino contiene almeno un arco.
Un grafo che non contiene cicli è detto **aciclico**.

Grafo connesso: un grafo non orientato è connesso se ogni coppia di vertici è collegata attraverso

almeno un cammino.
Le **componenti connesse** di un grafo sono le classi di equivalenza della relazione *è raggiungibile da*.
Un grafo non orientato è connesso se ha esattamente una componente connessa (ogni vertice è raggiungibile da ogni altro vertice).

Grafo fortemente connesso: un grafo orientato è fortemente connesso se due vertici qualsiasi sono raggiungibili l'uno dall'altro.
Le componenti fortemente connesse di un grafo orientato sono le classi di equaivalenza dei vertici secondo la relazione *sono mutuamente raggiungibili*.
Un grafo orientato è fortemente connesso se ha una sola componente connessa.

Grafi speciali:

- **Grafo completo**: grafo non orientato in cui ogni coppia di vertici è adiacente.
- **Foresta**: un grafo aciclico e non orientato.
- **Grafo sparso**: grafi in cui $|E| <<< |V|^2$
- **Grafo denso**: grafi in cui $|E| \sim |V|^2$
- **Grafo pesato**: grafi per cui ogni arco ha un peso associato, cioè una funzione $w : E \mapsto \mathbb{R}$

8.2 Rappresentazione mediante liste di adiacenza

L'uso di liste di adiacenza per rappresentare un grafo è particolarmente adatta per i **grafi sparsi**, ed è basata sulla creazione di un:
Array di adiacenza: array Adj di $|V|$ liste, una per ogni vertice v. Ciascuna lista contiene tutti i vertici raggiungibili da v.

Se G è un grafo pesato, il peso $w(u, v)$ dell'arco (u, v) viene memorizzato insieme al vertice v nella lista di adiacenza di u.

Se G è orientato, la somma delle lunghezze delle liste di adiacenza è $|E|$.
$\rightarrow (u, v)$ è rappresentato inserendo v in $Adj[u]$.
Se G è non orientato, la somma delle lunghezze delle liste di adiacenza è $2|E|$.
$\rightarrow (u, v)$ appare come inserimento di u in $Adj[v]$ e viceversa.
La **quantità di memoria** richiesta per rappresentare un grafo (orientato o non orientato) con le liste di adiacenza è $\Theta(|V| + |E|)$.

8.3 Rappresentazione mediante matrice di adiacenza

Questa rappresentazione è adatta per i **grafi densi**, oppure nelle applicazioni in cui è utile saper stabilire rapidamente se esiste un arco che collega due vertici particolari.

Si suppone che i vertici del grafo siano numerati $1, ..., |V|$ in modo arbitrario. La matrice di adiacenza $A = (a_{ij})$ di dimensione $|V| \times |V|$ è costruita come:

$$(a_{ij}) = \begin{cases} 1 & \text{se esiste un arco } (i, j) \\ 0 & \text{altrimenti} \end{cases} \tag{8.3}$$

Se G è un grafo pesato, con funzione peso $w(i, j)$:

$$(a_{ij}) = \begin{cases} w(i, j) & \text{se esiste un arco } (i, j) \\ 0 & \text{altrimenti} \end{cases} \tag{8.4}$$

La **quantità di memoria** richiesta per la rappresentazione di un grafo mediante matrice di adiacenza è $\Theta(V^2)$, indipendentemente dal numero di archi nel grafo.

8.4 Visita BFS (Breadth-First Search)

Dato un grafo $G = (V, E)$ e un vertice distinto s (detto **sorgente**), la visita in ampiezza ispeziona gli archi di G per scoprire tutti i vertici raggiungibili da s.

La visita in ampiezza è chiamata così perché **espande la frontiera** lungo l'ampiezza, ovvero, l'algoritmo scopre tutti i vertici che distano k da s prima di scoprire quelli distanti $k + 1$.

La BFS genera un **albero BF**, con radice s, che contiene tutti i vertici raggiungibili: per ogni vertice v raggiungibile da s, il cammino semplice che va da s a v corrisponde a un cammino minimo da s a v.

Per tenere traccia dei vertici visitati, la visita in ampiezza **colora** i vertici:

- **Bianco**: i vertici non ancora scoperti sono tutti bianchi.
- **Nero**: tutti i vertici adiacenti ad un vertice nero sono stati scoperti.
- **Grigio**: i vertici grigi sono stati scoperti, ma non sono ancora stati analizzati tutti i vertici ad esso adiacenti. Rappresentano la frontiera tra vertici scoperti e vertici da scopire.

```
1   BFS (G, s)
        //Coloriamo di bianco tutti i vertici tranne s
3       for (each u ∈ (G.V − {s}))
            u.color = WHITE;
5           u.d = ∞
            u.π = NULL
7       //Impostiamo gli attributi di s
        s.color = GRAY
9       s.d = 0
        s.π = NIL
11      //Creiamo una coda vuota e ci inseriamo s
        Q = 0
13      ENQUEUE (Q, s)
        while (Q ≠ 0)
15          //Estraiamo il primo vertice dalla coda
            u = DEQUEUE (Q)
17          //Per ogni vertice bianco adiacente a u, ne impostiamo i
    parametri e lo aggiungiamo alla coda.
            for (each v ∈ G.Adj[u])
19              if (v.color == WHITE)
                    v.color = GREY
21                  v.d = u.d + 1
                    v.π = u
23              ENQUEUE(Q, v)
            u.color = BLACK
25
```

Listing 8.1: BFS

I risultati dell'algoritmo BFS possono dipendere dall'ordine in cui i vicini di un dato vertice vengono visitati: l'albero può variare, ma le distanze calcolate dall'algoritmo sono le stesse.

8.5 Visita DFS (Depth-First Search)

8.5.1 L'idea

La strategia DFS consiste nell'esplorazione del grafo in **profondità**:

- Gli archi vengono ispezionati a partire dall'ultimo vertice scoperto v che ha ancora archi uscenti non ispezionati.
- Quando tutti gli archi di v sono stati ispezionati, la visita *torna indietro* per esplorare gli archi uscenti dal vertice dal quale v era stato scoperto.
- Questo processo continua finché non saranno stati scoperti tutti i vertici che sono raggiungibili dal vertice sorgente originale.
- Se restano dei vertici non scoperti, uno di questi viene selezionato come nuovo vertice sorgente e la visita riparte da questa sorgente.
- L'intero processo viene ripetuto finché non vengono scoperti tutti i vertici del grafo.

8.5.2 Il sottografo dei predecessori

Come nella visita in ampiezza, nella visita in profondità assegnano a ciascun nuovo vertice scoperto il suo predecessore come genitore.
A differenza della BFS, che genera un albero, la DFS produce il **sottografo dei predecessori** in forma di foresta, detta **foresta DF** e indicata con $G_\pi(V, E_\pi)$:

$$E_\pi = \{(v.\pi, v) : v \in V \wedge v.\pi \neq NIL\} \tag{8.5}$$

Gli archi di E_π sono detti **archi d'albero**.

8.5.3 Colorazione dei vertici

Per tenere traccia dei vertici visitati, la visita in profondità **colora** i vertici in maniera assoltamente analoga alla visita in ampiezza:

- **Bianco**: i vertici non ancora scoperti sono tutti bianchi.
- **Nero**: tutti i vertici adiacenti ad un vertice nero sono stati scoperti.
- **Grigio**: i vertici grigi sono stati scoperti, ma non sono ancora stati analizzati tutti i vertici ad esso adiacenti. Rappresentano la frontiera tra vertici scoperti e vertici da scopire.

8.5.4 Temporizzazione della DFS

Con la DFS associamo a ciascun vertice v due attributi temporali:

- $v.d$ (momento in cui il vertice viene scoperto): v diventa GRIGIO
- $v.f$ (momento in cui l'algoritmo finisce di esplorare il vertice): v diventa NERO

I tempi di scoperta dei vertici hanno una **struttura a parentesi**: se rappresentiamo $u.d$ con (u e $u.f$ con u) otteniamo un'espressione in cui le parentesi sono ben annidate.
Inoltre, per il **teorema delle parentesi** si verifica, per ogni coppia di vertici (u, v) una e una sola delle seguenti condizioni:

- $[u.d, u.f] \cap [v.d, v.f] = \emptyset$
  u e v non sono discendenti l'uno dall'altro nella foresta DF.

- $[u.d, u.f] \subset [v.d, v.f]$
  u è un discendente di v nella foresta DF.
- $[v.d, v.f] \subset [u.d, u.f]$
  v è un discendente di u nella foresta DF.

In particolare, si ha che v è un discendente proprio del vertice u nella foresta DF se e soltanto se:

$$u.d < v.d < v.f < u.f \tag{8.6}$$

8.5.5 Teorema del cammino bianco

In una foresta DF di un grafo $G = (V, E)$ (orientato o non orientato), il vertice v è un discendente del vertice u se e soltanto se, al tempo $u.d$ in cui viene scoperto u, il vertice v può essere raggiunto da u lungo un cammino che è formato esclusivamente da vertici bianchi.

8.5.6 Classificazione degli archi

Possiamo usare la DFS per classificare gli archi del grafo $G = (V, E)$ come segue:

- **Archi d'albero**: sono gli archi nella foresta DF. L'arco (u, v) è un arco d'albero se e solo se v viene scoperto per la prima volta da u.
- **Archi all'indietro**: sono quegli archi (u, v) che collegano un vertice u a un antenato v in un albero DF. I cappi, che possono presentarsi nei grafi orientati, sono considerati archi all'indietro.
- **Archi in avanti**: sono gli archi (u, v), diversi dagli archi d'albero, che collegano un vertice u a un suo discendente v nella foresta DF.
- **Archi trasversali**: tutti gli altri archi. Possono connettere i vertici nello stesso albero DF, purché un vertice non sia antenato dell'altro, oppure possono connettere vertice di alberi DF differenti.

Tramite la DFS, possiamo classificare ogni arco (u, v) in base al colore del vertice v che incontra quando l'arco viene ispezionato per la prima volta:

- $WHITE$: arco d'albero
- $GRAY$: arco all'indietro
- $BLACK$: arco in avanti o trasversale

In una DFS di un grafo non orientato, possono presentarsi solo archi d'albero o archi all'indietro. **NON** archi trasversali o in avanti.

8.5.7 L'algoritmo

```
DFS(G)
    for each u ∈ V
        u.color = WHITE
        u.π = NIL
    time = 0
    //Esploriamo i vertici bianchi
    for each u ∈ V
        if (u.color == WHITE)
            DFS-Visit (G, u)
//Visita del vertice u
```

```
    DFS-Visit (G, u)
12      time = time + 1
        u.d = time
14      //Il vertice u e' appena stato scoperto
        u.color = GRAY
16      for each v ∈ Adj[u]
            //Ispezioniamo l'arco (u,v)
18          if (v.color == WHITE)
                v.π = u
20              DFS-Visit(G, v)
        //Abbiamo completato la visita di u
22      u.color = BLACK
        time = time+1
24      u.f = time
```

Listing 8.2: DFS

8.6 Ordinamento topologico

Possiamo sfruttare la DFS per eseguire l'ordinamento topologico di un grafo diretto aciclico (**dag**).
Un ordinamento topologico di un dag è un ordinamento lineare di tutti i suoi vertici $v \in V$ tale che, se G contiene un arco (u, v), allora u appare prima di v nell'ordinamento.

Se il grafo non è aciclico, non è possibile effettuare alcun ordinamento topologico.

Possiamo vedere l'ordinamento topologico come un ordinamento dei vertici del grafo lungo una linea orizzontale in modo che tutti gli archi siano orientati da sinistra a destra.

```
    TopologicalSort (G)
2       Chiama DFS per calcolare tempi v.f
            per tutti i vertici
4       Completata ispezione di un vertice
            aggiungilo in testa a una lista
6       return la lista concatenata
```

La complessità dell'ordinamento topologico è $\Theta(|V| + |E|)$, perché la visita in profondità impiega un tempo $\Theta(|V| + |E|)$ e il tempo necessario per inserire ciascuno dei vertici in testa alla lista è $O(1)$.

Un grafo G è aciclico se e solo se la DFS di G non genera archi all'indietro.

8.7 Componenti fortemente connesse

```
    Strongly-Connected-Components(G)
2       Chiama DFS per calcolare tempi v.f
            per tutti i vertici
4       Calcola G^T
        Chiama DFS ma nel ciclo principale
6           considera i vertici in ordine decrescente
```

```
        rispetto ai tempi u.f appena calcolati
    Genera dei vertici di ciascun albero
        della foresta DF prodotta come
        singola componente fortemente connessa
```

Listing 8.3: Strongly Connected Components

Siano C e C' due cfc distinte con $u, v \in C$ e $u', v' \in C'$: se $u \to u'$, allora non può essere $v \to v'$.
Siano C e C' due cfc distinte, con $u \in C$ e $v \in C'$:

$$\begin{aligned} (u,v) \in E &\quad \to \quad C.f > C'.f \\ (u,v) \in E^T &\quad \to \quad C.f < C'.f \end{aligned} \tag{8.7}$$

L'algoritmo Strongly-Connected-Components ha complessità $\Theta(|V| + |E|)$

8.8 Alberi di connessione minimi

Consideriamo un grafo, orientato e pesato, $G = (V, E)$.
Ci interessa trovare un sottoinsieme aciclico $T \subseteq E$ che collega tutti i vertici minimizzando il peso:

$$w(T) = \sum_{(u,v) \in T} w(u,v) \tag{8.8}$$

Poiché T è aciclico, e collega tutti i vertici, deve anche formare un albero.
T è detto **albero di connessione minimo** (MST, Minimum Spanning Tree).

I due algoritmi che ci permettono di trovare $MST(G)$ sono quelli di **Kruskal** e **Prim**. Entrambi sono **algoritmi greedy**: la strategia greedy ("golosa") prevede una scelta fra diverse possibilità, che viene effettuata scegliendo la soluzione che in quel particolare momento è ritenuta la migliore. In generale questa strategia non fornisce soluzioni sempre ottime, ma in questo caso funziona.

8.8.1 Metodo generico

Supponiamo di avere un grafo $G = (V, E)$ connesso non orientato. Per trovare l'albero di connessione minimo, lo facciamo crescere un arco alla volta. Il metodo generico gestisce un insieme di archi A, conservando la stessa invariante di ciclo:

Prima di ogni iterazione, A è un sottoinsieme di qualche MST

Ad ogni passo, aggiungiamo ad A un arco che può essere aggiunto senza violare l'invariante. Tale arco è detto **arco sicuro** per A e trovarlo è il vero problema (è nel metodo di ricerca di tale arco che i due algoritmi che vedremo differiscono.)

```
MST-Generic(G, w)
    A = 0
    while (A non forma un albero di connessione)
        trova un arco (u,v) sicuro per A
        A = A ∪ (u,v)
    return a
```

Un **taglio** $(S, V-S)$ è una partizione di V. Un arco $(u,v) \in E$ **attraversa** il taglio $(S, V-S)$ se una delle sue estremità si trova in S e l'altra in $V-S$.
Un taglio **rispetta** un insieme A di archi se nessun arco di A attraversa il taglio.
Un arco è un **arco leggero** per un taglio se il suo peso è minimo fra i pesi degli altri archi che attraversano il taglio.

Teorema del taglio: sia $G = (V,E)$ un grafo connesso non orientato con una funzione peso w a valori reali definita in E. Sia A un sottoinsieme di E che è contenuto in qualche albero di connessione minimo per G e sia $(S, V-S)$ un taglio qualsiasi di G che rispetta A. Allora l'arco è sicuro per A.

8.8.2 Algoritmo di Kruskal

L'algoritmo di Kruskal trova un arco sicuro da aggiungere alla foresta in costruzione scegliendo fra tutti gli archi che collegano due alberi qualsiasi nella foresta, un arco (u,v) di peso minimo.
L'algoritmo di Kruskal usa una UnionFind per mantenere vari insiemi disgiunti di elementi. Ogni insieme contiene i vertici di un albero della foresta corrente.
L'operazione `Find-Set(u)` restituisce un rappresentante dell'insieme che contiene u. Dunque, possiamo determinare se due vertici appartengono allo stesso albero verificando se `Find-Set(u)` è uguale a `Find-Set(v)`.
L'unione degli alberi viene effettuata mediante la procedura `Union(u,v)`.
Il tempo di esecuzione dell'algoritmo di Kruskal dipende dall'implementazione della `UnionFind`, usando quella già vista, possiamo ottenere un tempo $O(E \cdot \log V)$

```
MST-Kruskal(G, w)
2   A = 0
    for (each v ∈ G.V)
4       Make-Set(v)
    ordina gli archi di G.E in senso non decrescente rispetto a
    w
6   for (each (u,v) ∈ G.E preso in ordine non decrescente)
        if (Find-Set(u) ≠ Find-Set(v))
8           A = A ∪{(u,v)}
            Union(u,v)
10  return A
```

Listing 8.4: Algoritmo di Kruskal

8.8.3 Algoritmo di Prim

L'algoritmo di Prim è un caso speciale del metodo generico per gli alberi di connessione minimi, ed è molto simile all'agloritmo di Kruskal, con alcune importanti differenze.
In primo luogo, gli archi dell'insime A formano sempre un **albero singolo**.

L'albero inizia da un arbitrario vertice r, e si sviluppa fino a coprire tutti i vertici in $G.V$. Ad ogni passo viene aggiunto all'albero A un arco leggero che collega A ad un vertice isolato (che non sia estremo di qualche arco in A).
Per il teorema del taglio questa regola aggiunge soltanto gli archi che sono sicuri per A; cosicché, quando l'algoritmo termina, gli archi in A formano un albero di connessione minimo.
Questa strategia è golosa perché l'albero cresce includendo a ogni passo un arco che contribuisce con la quantità più piccola possibile a formare il peso dell'albero.

La chiave per implementare con efficienza l'algoritmo di Prim consiste nel trovare un modo veloce per

scegliere un nuovo arco da aggiungere all'albero formato dagli archi in A.

Il grafo connesso G e la radice r dell'albero di connessione minimo da costruire sono gli input dell'algoritmo.
Durante l'esecuzione dell'algoritmo, tutti i vertici che non si trovano nell'albero risiedono in una coda di min-proprietà Q basata su un campo key.
Per ogni vertice v, l'attributo $v.key$ è il peso minimo di un arco qualsiasi che collega v a un vertice nell'albero; per convenzione, $v.key = \infty$ se tale arco non esiste.
L'attributo $v.\pi$ indica il padre di v nell'albero. Durante l'esecuzione dell'algoritmo, l'insieme A di `Generic-MST` è mantenuto implicitamente come

$$A = (v, v.\pi) : v \in V - \{r\} - Q \tag{8.9}$$

Quando l'algoritmo termina, la coda di min-priorità Q è vuota; l'albero di connessione minimo A per G è quindi

$$A = (v, v.\pi) : v \in V - \{r\} \tag{8.10}$$

```
        MST-Prim (G, w, r)
2           for (each u ∈ G.V)
                u.key = ∞
4               u.π = NIL
            r.key = 0
6           Q = G.V
            while (Q ≠ 0)
8               //Identifica un vertice incidente su un arco leggero che
    attraveresa il taglio (V-Q, Q)
                u = Extract-Min(Q)
10              //Aggiorna i campi key e parent dei vertici adiacenti a
    u che non appartengono all'albero
                for (each v ∈ G.Adj[u])
12                  if (v ∈ Q and w(u,v) < v.key)
                        v.π = u
14                      v.key = w(u,v)
```

Listing 8.5: Algoritmi di Prim

L'algoritmo conserva la seguente invariante di ciclo composta da tre parti:

1. $A = (v, v.\pi) : v \in V - \{r\} - Q$
2. I vertici già inseriti nell'albero di connessione minimo sono quelli che appartengono a $(V - Q)$
3. Per ogni vertice $v \in Q$, se $v.\pi = NIL$, allora $v.key < \infty$ e $v.key$ è il peso di un arco leggero $(v, v.\pi)$ che collega v a qualche vertice che si trova già nell'albero di connessione minimo.

Le prestazioni dell'algoritmo di Prim dipendono dall'implementazione della coda Q. Se implementiamo Q come un heap minimo binario, ppossiamo ottenere un tempo totale $O(E \log V)$, che è asintoticamente uguale a quello dell'algoritmo di Kruskal.

8.9 Cammini minimi

8.9.1 Definizioni e proprietà

In un **problema dei cammini minimi** è dato un grafo orientato pesato $G = (V, E)$ con una funzione peso $w : E \to \mathbb{R}$, che associa agli archi dei pesi di valore reale.

Il **peso** $w(p)$ del cammino $p =< v_0, ..., v_k >$ è la somma dei pesi degli archi che lo compongono:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{8.11}$$

Il **peso di un cammino minimo** $\delta(u, v)$ da u a v è definito come segue:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{se esiste un cammino da u a v} \\ \infty & \text{negli altri casi} \end{cases} \tag{8.12}$$

Un **cammino minimo** dal vertice u al vertice v è definito come un cammino qualsiasi p con peso $w(p) = \delta(u, v)$

8.9.2 Rappresentazione dei cammini minimi

Spesso è necessario calcolare non soltanto i pesi dei cammini minimi, ma anche i vertici in questi cammini. La rappresentazione che utilizziamo per i cammini minimi è simile a quella utilizzata per gli alberi delle visite in ampiezza.
Dato un grafo $G = (V, E)$ manteniamo per ogni vertice $v \in V$ un **predecessore** $v.\pi$ che può essere un altro vertice o un valore NIL.
L'algoritmo di Dijkstra impostano gli attributi π in modo che la catena dei predecessori che inizia da un vertice v percorra all'indietro un cammino minimo da s a v.

Durante l'esecuzione di un algoritmo per cammini minimi, tuttavia, i valori π non devono necessariamente indicare i cammini minimi. Come nella visita in ampiezza, saremo interessati al **sottografo dei predecessori** $G_\pi = (V_\pi, E_\pi)$ indotto dai valori π.
Definiamo V_π come l'insieme dei vertici di G con predecessori non NIL, più la sorgente s:

$$V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\} \tag{8.13}$$

L'insieme degli archi orientati E_π è l'insieme degli archi indotto dai valori π per i vertici in V_π:

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\} \tag{8.14}$$

L'algoritmo di Dijkstra ha la proprietà che G_π è un **albero dei cammini minimi** (un albero radicato che contiene un cammino minimo dalla sorgente s a ogni vertice che è raggiungibile de s).
Un albero di cammini minimi è simile all'albero di visita in ampiezza, con la differenza che contiene i cammini dalla sorgente che sono definiti minimi in base ai pesi degli archi, anziché al numero degli archi.

Sia $G = (V, E)$ un grafo orientato pesato con la funzione peso $w : E \to \mathbb{R}$; supponiamo che G non contenga cicli di peso negativo raggiungibili dalla sorgente $s \in V$, in modo che i cammini minimi siano ben definiti.
Un **albero di cammini minimi** radicato in s è un sottografo orientato $G' = (V', E')$ dove $V' \subseteq V$ e $E' \subseteq E$, tale che:

- V' è l'insieme dei vertici raggiunbili da s in G
- G' forma un albero con radice s
- Per ogni $v \in V'$ l'unico cammino semplice da s in v in G' è un cammino minimo da s a v in G

I cammini minimi non sono necessariamente unici, né lo sono gli alberi dei cammini minimi.

8.9.3 Rilassamento di un arco

L'algoritmo di Dijkstra utilizza la tecnica del **rilassamento**: per ogni vertice $v \in V$, manteniamo un attributo $v.d$ che è un limite superiore per il peso di un cammino minimo dalla sorgente s al vertice v. L'attributo $v.d$ è detto **stima del cammino minimo**.
La seguente procedura con tempo $\Theta(V)$ inizializza le stime dei cammini minimi e i predecessori.

```
Initialize-Single-Source(G, s)
    for (each v ∈ G.V)
        v.d = ∞
        v.π = NIL
    s.d = 0

```

Listing 8.6: Inizializzazione delle stime dei cammini minimi

Il processo di **rilassamento** di un arco (u, v) consiste nel verificare se, passando per u, è possibile migliorare il cammino minimo per v precedentemente trovato e, in caso affermativo, nell'aggiornare $v.d$ e $v.\pi$.
Un passo di rilassamento può ridurre il valore della stima del cammino minimo $v.d$ e aggiornare il campo $v.\pi$ del predecessore di v.
Il seguente codice effettua un passo di rilassamento sull'arco (u, v) in tempo $O(1)$.

```
Relax(u,v,w)
    if (v.d > u.d + w(u,v))
        v.d = u.d + w(u,v)
        v.π = u

```

Listing 8.7: Rilassamento di un arco

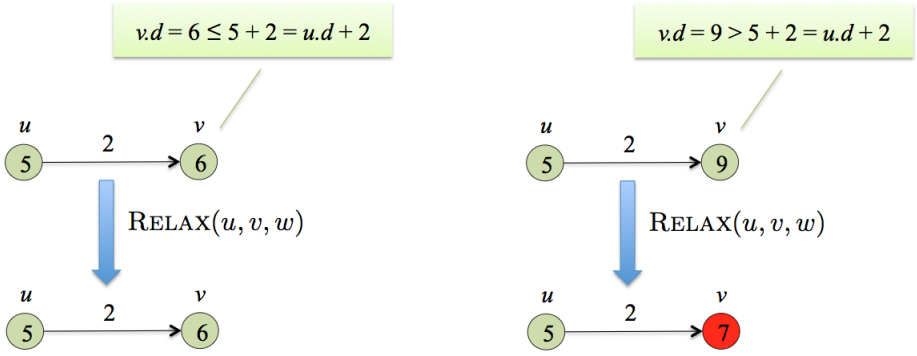

Figura 8.1: Rilassamento di archi

8.9.4 Condizione di Bellman

Per qualsiasi arco $(u, v) \in E$ si ha:

$$\delta(s, v) \leq \delta(s, u) + w(u, v) \tag{8.15}$$

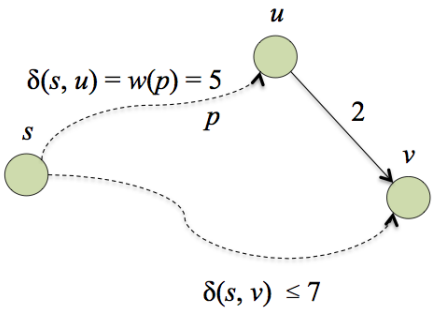

Figura 8.2: Condizione di Bellman

8.9.5 Algoritmo di Dijkstra

L'algoritmo di Dijkstra risolve il problema dei cammini minimi da sorgente unica in un grafo orientato pesato $G = (V, E)$ nel caso in cui tutti i pesi degli archi non siano negativi.
L'algoritmo di Dijkstra mantiene un insieme S di vertici i cui pesi finali dei cammini minimi della sorgente s sono stati già determinati. L'algoritmo seleziona ripetutamente il vertice $u \in (V - S)$ con la stima minima del cammino minimo, aggiunge $u \in S$ e rilassa tutti gli archi che escono da u.

Nella nostra implementazione manteniamo una coda di min-priorità Q di vertici, utilizzando come chiavi i loro valori d.

```
Dijkstra (G, w, s)
    Initialize-Single-Source (G, s)
    S = 0
    Q = G.V
    while (Q ≠ 0)
        u = Extract-Min(Q)
        S = S ∪{ S }
        for (each v ∈ G.Adj[u])
            Relax (u, v, w)
```

Listing 8.8: Algoritmo di Dijkstra