Indice

Algoritmi

1 Grafi come strutture dati

1.1 Introduzione e terminologia

Un grafo è una coppia di elementi (insiemi) **G=(V,E)** e consiste in:

- un insieme V di **vertici** (o **nodi**)
- un insieme E (E sottoinsieme del prodotto cartesiano $V \times V$) di coppie di vertici, detti **archi** o **spigoli**; ogni arco connette due vertici

I grafi possono essere:

- **orientati**: relazioni asimmetriche, insieme di coppie ordinate
- **non orientati**: relazioni simmetriche, insieme di coppie non ordinate

Un arco è **incidente** per i nodi che si toccano.
Il **grado** di un vertice è dato dal numero di archi incidenti.
Un vertice B è **adiacente** ad A se da B si può percorrere un solo arco e giungere ad A.
Un **sottografo** è una porzione di grafo (notazione $H \subseteq G$): i vertici di H sono sottoinsieme dei vertici di G e gli archi di H sono sottoinsieme degli archi di G.
Un **cammino** è una sequenza ordinata di archi che collegano due nodi. I cammini devono rispettare l'orientamento degli archi. La **lunghezza** è il numero di archi di cui è composto un cammino.
Un cammino si dice **semplice** se non passa due volte per lo stesso vertice. Se esiste almeno un cammino p tra i vertici v e w, si dice che w è **raggiungibile** da v. Inoltre v è un **antenato** di w e w è un **discendente** di v.
Un cammino tra due nodi v e w si dice **minimo** se tra v e w non esiste nessun altro cammino di lunghezza minore. La lunghezza del cammino minimo è detta **distanza** ($\delta(v, w)$).
Un grafo può essere **pesato**. La funzione peso è definita come $W : E \to \mathbb{R}$; per ogni arco $(v, w) \in E, W(v, w)$ definisce il **peso** di (v, w). In un grafo pesato, la lunghezza/peso di un cammino si calcola sommando i pesi degli archi che contiene.
I grafi non orientati possono essere:

- **connessi**: esiste un cammino da ogni vertice verso ogni altro vertice
- **non connessi**

I grafi orientati possono essere:

- **fortemente connessi**: esiste un cammino da ogni vertice verso ogni altro vertice
- **debolmente connessi**: ignorando il verso degli archi

Un cammino $\langle w_1, w_2, ..., w_n \rangle$ si dice **chiuso** se $w_1 = w_n$. Un cammino chiuso, semplice, di lunghezza almeno 1 si dice **ciclo**. Se un grafo non contiene cicli, si dice **aciclico**.
Un **grafo completo** è un grafo con un arco per ogni coppia di vertici. Un grafo completo ha numero di archi E pari a $|E| = \frac{|V|(|V|-1)}{2}$.
Un grafo non orientato, connesso e aciclico è definito **albero libero**. Se un vertice è designato ad essere radice, si definisce **albero radicato**. Un grafo non orientato, aciclico ma non connesso è definito **foresta**.

1.2 Rappresentazione

Per valutare un approccio di rapppresentazione, bisogna considerare lo **spazio** occupato dalla struttura dati e il **costo computazionale** delle operazioni da effettuare su di essa.

1.2.1 Lista di archi

Dati n (numero di vertici) e m (numero di archi), lo spazio occupato è $\mathcal{O}(n+m)$: è una rappresentazione inefficiente, in quanto bisogna percorrere tutto il grafo per scandire la lista di archi. Introdurre un vertice o arco ha costo $\mathcal{O}(1)$, ma la rimozione ha costo $\mathcal{O}(m)$.

1.2.2 Liste di adiacenza

Ogni vertice v ha una lista contenente i vertici ad esso adiacenti. Calcolare il grado di un vertice è un'operazione semplice, in quanto basta scorrere la lista di adiacenza. Occupa spazio $\mathcal{O}(n+m)$, ed è adatta per grafi **sparsi** (il numero di archi è molto minore del numero di vertici).

1.2.3 Liste di incidenza

Ogni vertice v ha una lista contenente un riferimento agli archi ad esso incidenti. Occupa spazio $\mathcal{O}(n+m)$.

1.2.4 Matrici di adiacenza

Il grafo è rappresentato tramite una matrice di interi di grandezza $n \times n$ (spazio occupato $\mathcal{O}(n^2)$); è adatta per grafi **densi**. Calcolare il grado e archi incidenti ha costo $\mathcal{O}(n)$ (basta scorrere la matrice). La modifica dei vertici ha costo $\mathcal{O}(n^2)$ in quanto bisogna ricostruire completamente la matrice. Una matrice di adiacenza rappresenta anche la presenza di un cammino di lunghezza 1 tra ogni coppia di vertici v e w. In particolare, $v \to_1 w$ se e solo se $M[v,w] \neq 0$: moltiplicando la matrice per sè stessa, il risultato è diverso da 0 solo se esiste un cammino di lunghezza 2 (e via dicendo).

1.2.5 Matrici di incidenza

Il grafo è rappresentato tramite una matrice di interi di grandezza $n \times m$ (spazio occupato $\mathcal{O}(n \times m)$), in cui le righe indicizzano i vertici e le colonne indicizzano gli archi.

2 Visite

2.1 Visita generica

Una **visita** di un grafo G permette di esaminare i nodi e gli archi in maniera sistematica, senza passare due volte per lo stesso nodo.

2.1.1 Inizializzazione

Una tattica per evitare di visitare un nodo più volte è quella di mappare lo stato della visita ad un colore:

- **bianco** (o **nodi inesplorati**): vertice non ancora esplorato
- **grigio** (o **nodi aperti**): vertice visitato, ma con nodi adiacenti ancora inesplorati
- **nero** (o **nodi chiusi**): vertice visitato, con adiacenti esplorati

Dati n nodi, si utilizza un vettore *color* di colori, di grandezza n: all'inizio della visita, tutte le celle del vettore *color* sono impostate a *white*.

Algoritmo 1 INIZIALIZZA(G)

color ← vettore di lunghezza n
for ogni $u \in V$ **do**

color[u] ← white
end for

La visita parte da un nodo s, detto **nodo sorgente**.

Algoritmo 2 VISITA(G,s)

```
INIZIALIZZA(G)
color ← gray
{visita s}
while ci sono vertici grigi do
    u ← scegli un vertice grigio
    if esiste v bianco adiacente ad u then
        color[v] ← gray
        {visita v}
    else color[v] ← black
    end if
end while
```

Il cambiamento di colore è **monotono** (bianco → grigio → nero).

2.1.2 Invarianti

Un'**invariante** è una condizione che è verificabile come vera sia all'inizio sia alla fine di un ciclo:

- Invariante 1: se esiste un arco $(u, v) \in E$ ed u è nero, allora v è grigio o nero
- Invariante 2: tutti i vertici grigi o neri sono raggiungibili dalla sorgente
- Invariante 3: qualunque cammino dalla sorgente ad un vertice bianco deve contenere almeno un vertice grigio

Teorema. *Al termine dell'algoritmo di visita, v è nero se e solo se v è raggiungibile dalla sorgente.*

Dimostrazione. Per l'invariante 2, all'uscita dal ciclo tutti i vertici neri sono raggiungibili da s. Dall'invariante 3 si ricava che tra s e v esiste almeno un vertice grigio, oppure v non è bianco. Dato che la condizione di uscita dal ciclo è quella che non esistano più vertici grigi, si ricava che v non è bianco (cambiamento monotono) e non può essere grigio. Quindi, all'uscita dal ciclo, tutti i vertici raggiungibili dalla sorgente sono neri. □

2.1.3 Predecessori

L'algoritmo può essere modificato in modo da ricordare, per ogni vertice che viene scoperto, quale vertice grigio ha permesso di scoprirlo, ossia ricordare l'arco percorso. Ad ogni vertice u si associa un attributo $\pi[u]$ che rappresenta il vertice che ha permesso di scoprirlo.

Algoritmo 3 VISITA(G,s)

```
INIZIALIZZA(G)
color ← gray
{visita s}
while ci sono vertici grigi do
```

```
    u ← scegli un vertice grigio
    if esiste v bianco adiacente ad u then
        color[v] ← gray
        π[v] ← u
        {visita v}
    else color[v] ← black
    end if
  end while
```

Proprietà. *Al termine dell'esecuzione di VISITA(G,s), tutti e soli i vertici neri diversi da s hanno predecessore diverso da NULL.*

Il sottografo dei predecessori è un albero (**albero dei predecessori**) di radice s.
Se il grafo non è connesso:

Algoritmo 4 VISITA TUTTI I VERTICI(G)

```
  INIZIALIZZA(G)
  for ogni u ∈ V do
    if color[u] = white then
      VISITA(G,u)
    end if
  end for
```

2.1.4 Gestione dei vertici grigi

Per gestire i nodi grigi si usa una struttura dati ordinata D (**frangia**). Sulla frangia è possibile eseguire le seguenti operazioni:

- **Create()**: restituisce una D vuota
- **Add(D,x)**: aggiunge un elemento x a D
- **First(D)**: restituisce il primo elemento di D
- **RemoveFirst(D)**: elimina il primo elemento di D
- **NotEmpty(D)**: restituisce vero se D contiene almeno un elemento, falso altrimenti

D è una **coda** se Add(D,x) aggiunge l'elemento in coda a D, uno **stack** se Add(D,x) aggiunge l'elemento in testa a D.

Algoritmo 5 VISITA(G,s)

```
  INIZIALIZZA(G)
  Create()
  color[s] ← gray
  {visita s}
  Add(D,s)
  while NotEmpty(D) do
    u ← First(D)
    if esiste v bianco adiacente ad u then
      color[v] ← gray
```

```
        π[v] ← u
        {visita v}
        Add(D,v)
    else
        color[v] ← black
        RemoveFirst(D)
    end if
end while
```

2.1.5 Complessità

Il costo di visita è $\mathcal{O}(n+adj)$: adj è il tempo impiegato a controllare se esiste un nodo v bianco adiacente ad u, e dipende dalla rappresentazione; n è il numero di vertici, che vengono inseriti e rimossi da D.
Il costo di adj è:

- con lista di archi: bisogna scandire l'intera lista ($\mathcal{O}(m)$) per n volte ($\mathcal{O}(n)$), quindi $\mathcal{O}(n)+\mathcal{O}(n*m) = \mathcal{O}(mn)$
- con matrice di adiacenza: bisogna scandire l'intera riga della matrice ($\mathcal{O}(n)$), quindi $\mathcal{O}(n) + \mathcal{O}(n * n) = \mathcal{O}(n^2)$
- con liste di adiacenza: si possono ottimizzare le prestazioni utilizzando dei puntatori che puntano all'inizio delle liste di adiacenza. Se l'elemento è grigio, il puntatore è spostato all'elemento successivo; quando il puntatore giunge alla fine della lista, il primo elemento è colorato di nero. Ogni lista è percorsa una volta sola, in tutte le iterazioni del ciclo. Complessità: $\mathcal{O}(n + m)$.

2.2 Visita in ampiezza

2.2.1 Inizializzazione

La **visita in ampiezza** (**BFS**, Breadth First Search), esamina i vertici del grafo in un ordine ben preciso, costruendo un albero di visita chiamato **albero BFS**. Nell'albero BFS, ogni vertice si trova il più vicino possibile alla radice. La visita è realizzata usando la frangia come coda: quando un nodo grigio ha tutti gli adiacenti grigi, esso è rimosso dalla cosa (il vertice in testa rimane nella coda finchè non diventa nero).

Algoritmo 6 VISITA BFS(G,s)

```
INIZIALIZZA(G)
queue()
color[s] ← gray
{visita s}
enqueue(D,s)
while NotEmpty(D) do
    u ← head(D)
    if esiste v bianco adiacente ad u then
        color[v] ← gray
        π[v] ← u
        {visita v}
        enqueue(D,v)
```

```
    else
        color[v] ← black
        dequeue(D)
    end if
end while
```

2.2.2 Albero di visita

L'albero BFS viene costruito a livelli; l'albero rappresenta i cammini minimi. Anche se le liste di adiacenza vengono invertite, i nodi per livello non cambiano. Si può inizializzare un **vettore di distanze** (stimate) d, inizializzato ad infinito: se un determinato vertice non è stato trovato (distanza ∞ a fine BFS), allora non è raggiungibile da s.

2.2.3 Proprietà

Proprietà (1)**.** *In D ci sono tutti e soli i vertici grigi.*

Proprietà (2)**.** *Se $\langle v_1, v_2, \ldots, v_n \rangle$ è il contenuto di D, allora:*

i $d[v_i] \leq d[v_{i+1}]$*: i vertici sono ordinati per livelli nella coda*

ii $d[v_n] \leq d[v_1] + 1$*: la coda contiene al massimo due livelli*

Dimostrazione. Nel caso base, in D è presente solo la sorgente. La proprietà 2 è vera.
Il passo ha due casi:

- dequeue(D): o D rimane vuota (banalmente vera), o rimangono $\langle v_2, \ldots, v_n \rangle$, e
  i. le disuguaglianze sono ancora vere e quindi
  ii. anche $d[v_n] \leq d[v_1] + 1 \leq d[v_2] + 1$
- enqueue(D,v): v è reso figlio di v_1 e accodato, quindi $d[v] = d[v_1] + 1$ e
  i. $d[v_n] \leq d[v_1] + 1 = d[v]$
  ii. $d[v] = d[v_1] + 1 \leq d[v_1] + 1$

□

2.2.4 Dimostrazione $d[v] = \delta(s, v)$

Lemma (Invariante 4)**.** $d[v] = \delta(s, v)$ *per tutti i vertici grigi o neri.*

Dimostrazione $d[v] \geq \delta(s, v)$**.** Dato che l'albero dei predecessori π contiene solo archi appartenenti a G, il cammino da s a v è un cammino che appartiene anche a G, quindi la lunghezza del cammino da s a v nell'albero è maggiore o uguale alla distanza tra s e v.

Dimostrazione $d[v] \leq \delta(s, v)$**.** Definiamo l'insieme dei vertici a distanza k dalla sorgente nel grafo come $V_k = v \in V | \delta(s, v) = k$ (v_0 contiene solo la sorgente).
Nel caso base, $d[v_0] \leq \delta(s, v)$ (distanza di s da sè stesso: $0 \leq 0$). Sia $v \in V_k$: allora $\delta(s, v) = k$ (per definizione).
Con $k > 0$ (passo), esisterà almeno un vertice w tale che $\delta(s, w) = k - 1$ e $(w, v) \in E$, ovvero un arco che va da v a w. Definiamo l'insieme dei vertici appartenenti a V_{k-1} con arco entrante in v come $U_{k-1} = w \in V_{k-1} | (w, v) \in E$. Tra questi, sia u il primo vertice di U_{k-1} ad essere scoperto ed inserito nella coda: per politica FIFO, u sarà anche il primo ad essere estratto dalla coda. Quando guarderò i vertici adiacenti a u, v sarà ancora bianco (perchè più lontano), e v verrà inserito nell'albero come figlio di u,

con $d[v] = d[u] + 1$. Inoltre, per ipotesi induttiva, $d[u] \leq k - 1$.
Quindi, quando inseriremo v nell'albero:

- $d[v] = d[u] + 1$
- ma $d[u] \leq k - 1$, quindi $d[v] \leq (k - 1) + 1$
- $d[v] \leq k$

2.2.5 Distanza nell'albero BFS

Teorema. *Al termine dell'esecuzione della visita BFS, si ha $d[v] = \delta(s, v)$ per tutti i vertici $v \in V$.*

Dimostrazione. caso base: se v non è raggiungibile da s, allora $d[v]$ rimane ∞.
altrimenti, v è nero (per invariante 2). Per ogni vertice v raggiungibile da s, il cammino da s a v sull'albero ottenuto dalla visita è un **cammino minimo**. □

2.3 Visita in profondità

2.3.1 Inizializzazione

La **visita in profondità** (**DFS**, Depth First Search), esamina i vertici del grafo partendo dall'ultimo vertice incontrato. Si ottiene dall'algoritmo di visita generico, implementando la frangia come **stack**.

Algoritmo 7 VISITA DFS (ottimizzata)

```
D ← emptyStack(D)
color[s] ← gray
{visita s}
push(D)
while esiste v non considerato adiacente a top(D) do
    if color[v] = white then
        color[v] = gray
        π[v] ← top(D)
        {visita s}
        push(D,s)
    end if
    color[top(D)] ← black
    pop(D)
end while
```

2.3.2 Caratteristiche dell'albero DFS

Un vertice viene chiuso (colorato di nero) solo quando tutti i suoi discendenti sono stati chiusi.
Gli intervalli di attivazione di una qualunque coppia di vertici sono o **disgiunti** o **uno contenuto interamente nell'altro**. Questo è l'ordine delle attivazioni delle chiamate di una procedura ricorsiva, ed è quindi possibile costruire un algoritmo DFS ricorsivo. Inoltre, è possibile introdurre un contatore per ricordare l'ordine delle attivazioni.

Algoritmo 8 VISITA DFS RICORSIVA(G,u)

```
color[u] ← gray
{visita s}
```

```
d[u] ← time
time++
for ogni v adiacente ad u do
    if color[v] = white then
        π[v] ← u
        VISITA DFS RICORSIVA(G,v)
    end if
    color[u] ← black
    f[u] ← time
    time++
end for
```

2.3.3 Teorema delle parentesi e corollari

Teorema (Parentesi)**.** *In ogni visita DFS di un grafo, per ogni coppia di vertici u, v, una ed una sola delle seguenti condizioni è soddisfatta:*

- $d[u] < d[v] < f[v] < f[u]$ *ed u è un* ***antenato*** *di v in un albero della foresta DFS*
- $d[v] < d[u] < f[u] < f[v]$ *ed u è un* ***discendente*** *di v in un albero della foresta DFS*
- $d[u] < f[u] < d[v] < f[v]$ *e tra u e v non esiste relazione (non sono adiacenti)*

Teorema (Annidamento degli intervalli)**.** *Una visita DFS di un grafo colloca un vertice v come discendente proprio di un vertice u in un albero della foresta DFS se e solo se* $d[u] < d[v] < f[v] < f[u]$.

Teorema (Cammino bianco)**.** *In una foresta DFS, un vertice v è discendente del vertice u se e solo se al tempo d[u] v è raggiungibile da u con un cammino contenente solo vertici bianchi.*

La complessità della visita DFS è $(O)(m + n)$ (come la visita generica).

3 Aciclicità

È possibile verificare la presenza di un ciclo tramite una visita DFS.

3.1 Grafi orientati

Un arco $\langle u, v\rangle$ viene **percorso** quando si incontra v nella lista degli adiacenti ad u. Durante la DFS di un grafo orientato, ogni arco è percorso una volta sola. Definiamo:

- **Arco dell'albero**: arco inserito nella foresta DFS
- **Arco all'indietro**: arco che collega un vertice ad un suo antenato
- **Arco in avanti**: arco che collega un vertice ad un suo discendente
- **Arco di attraversamento**: arco che collega due vertici che non sono in relazione

Durante la visita di un grafo orientato, un arco $\langle u, v\rangle$ viene percorso quando si incontra v nella lista degli adiacenti ad u. In quel momento, color[v] può essere:

- **bianco**: $\langle u, v\rangle$ è un **arco dell'albero**

- **grigio**: u è un discendente di v, $\langle u, v\rangle$ è un **arco all'indietro**
- **nero**: $\langle u, v\rangle$ è un arco
  - **in avanti** se v è discendente di u
  - **di attraversamento** altrimenti

3.2 Grafi non orientati

Durante la DFS di un grafo orientato, ogni arco è percorso esattamente due volte. Definiamo:

- **Arco dell'albero**: arco inserito nella foresta DFS
- **Arco all'indietro**: arco che collega un vertice ad un suo antenato

Durante la visita di un grafo orientato, un arco $\langle u, v\rangle$ viene percorso quando si incontra v nella lista degli adiacenti ad u. In quel momento, color[v] può essere:

- **bianco**: (u, v) è un **arco dell'albero**
- **grigio**: u è un discendente di v, $\langle u, v\rangle$ è un **arco all'indietro**
- **nero**: $\langle u, v\rangle$ è un **arco all'indietro**

3.3 Test di aciclicità

Teorema (Grafo aciclico)**.** *Se un grafo (orientato o non orientato) contiene un ciclo, allora esiste un arco all'indietro. Viceversa, se una visita in profondità produce un arco all'indietro, il grafo contiene un ciclo. Quindi, un grafo è aciclico se e solo se una visita DFS non produce archi all'indietro.*

Algoritmo 9 CICLICO(G)

```
INIZIALIZZA(G)
for ogni nodo u di G do
    if color[u] = white and VISITA RICORSIVA CICLO(G,u) then
        return true
    end if
end for
return false
```

Algoritmo 10 VISITA RICORSIVA CICLO(G,u)

```
color[u] ← gray
for ogni v adiacente ad u do
    if color[v = white] then
        π[v] ← u
        if VISITA RICORSIVA CICLO(G,v) then
            return true
        end if
    else if v ≠ π[u] (color[v] = gray per grafi orientati) then
        return true
    end if
end for
color[u] ← black
return false
```

4 Ordinamento topologico

Dato un **grafo orientato aciclico** (DAG) è sempre possibile ordinare i nodi in un **ordine topologico**, cioè in modo che non ci sia nessun arco all'indietro nell'ordinamento.

4.1 Raggiungibilità

In un DAG, la relazione di **raggiungibilità** è una relazione di ordine parziale:

- è riflessiva: ogni vertice è raggiungibile da se stesso
- è antisimmetrica: se v è raggiungibile da u ed u è raggiugibile da v, allora v e u sono coincidenti
- è transitiva: se v è raggiungibile da u e w è raggiungibile da v, allora w è raggiungibile da u

4.2 Ordine topologico

Dato un DAG, un **ordine topologico** è un ordine lineare dei suoi nodi tale che, se nel grafo vi è un arco (u, v), allora u precede v nell'ordine. Un grafo aciclico possiede sempre un ordine topologico; un DAG può possedere diversi ordini topologici.

4.2.1 Algoritmo astratto

Il metodo più semplice ma meno efficiente per trovare un'ordinamento topologico. Si definiscono: **nodo sorgente** come nodo che non ha archi entranti, e **nodo pozzo** come nodo che non ha archi uscenti. Questo algoritmo funziona rimuovendo nodi sorgente dal grafo G' (memorizzandoli in una lista, *ord*) e sviluppando un sottoproblema.

Teorema. *ord è un'ordinamento topologico di G. Denotiamo con ord_i e G' (copia del grafo G) il contenuto di ord e G' all'iterazione i-esima: ad ogni istante del ciclo, non può esserci nessun cammino in G che porta da un vertice in G' ad uno in ord ("all'indietro").*

Dimostrazione. Caso base: con $i = 0$, la condizione è banalmente verificata (*ord* non contiene vertici). Passo induttivo: ad un istante k, non c'è alcun cammino in G dai vertici G'_k ai vertici in ord_k, quindi ad un istante $k + 1$, non c'è alcun cammino in G dai vertici G'_{k+1} ai vertici in ord_{k+1}.
Al passo k-esimo, si sceglie un vertice sorgente u in G'_k: non ci sono cammini in G che raggiungono u passando solo per i vertici di G'_k. Per ipotesi induttiva, aggiungendo u ad ord_k all'istante k, all'istante $k + 1$ non ci sarà alcun cammino in G dai vertici di G'_{k+1} ai vertici in ord_{k+1}. □

La complessità dell'algoritmo semplice è $\mathcal{O}(m * n)$.

4.2.2 Ordinamento topologico basato su DFS

Teorema (Ordinamento topologico)**.** *In una qualunque visita DFS, $f[v] < f[u]$ per ogni arco $\langle u, v\rangle$.*

Dimostrazione. Supponiamo per assurdo che si abbia $f[u] < f[v]$; quindi, apro e chiudo u e poi apro e chiudo v, oppure apro u e v e poi chiudo u e v. Il primo caso è impossibile, perchè u non può diventare nero prima che v diventi grigio, ossia prima che tutti i suoi adiacenti siano stati scoperti. Il secondo caso è impossibile perchè u sarebbe discendente di v e l'arco $\langle u, v\rangle$ sarebbe un arco all'indietro, ma il grafo è aciclico. □

Per ottenere l'ordinamento topologico, basta tenere traccia dei vertici chiusi con una lista, che conterrà i vertici chiusi, dal più al meno recente.

Algoritmo 11 TOPOLOGICAL SORT(G)

```
INIZIALIZZA(G)
ord ← vettore di lunghezza n
t ← n-1
for ogni u ∈ V do
    if color[u] = white then
        TOPOLOGICAL DFS(G,u,ord,t)
    end if
end for
return ord
```

Algoritmo 12 TOPOLOGICAL DFS(G,u,ord,t)

```
color[u] ← gray
d[u] ← time ← time+1
for ogni v adiacente ad u do
    if color[v] = white then
        πgetsu
        TOPOLOGICAL DFS(G,v,ord,t)
    end if
end for
color[u] ← black
f[u] ← time ← time+1
ord[t] ← u
t–
```

La complessità è $\mathcal{O}(m+n)$.

5 Componenti connesse e fortemente connesse

5.1 Componenti connesse

In un grafo non orientato, la relazione di raggiungibilità è una relazione di equivalenza:

- è **riflessiva**: per definizione, ogni vertice è raggiungibile da se stesso con un cammino degenere di lunghezza 0
- è **simmetrica**: se v è raggiungibile da u tramite un cammino p, allora u è raggiungibile da v percorrendo all'indietro lo stesso cammino
- è **transitiva**: se v è raggiungibile da u e w è raggiungibile da v, allora w è raggiungibile da u, percorrendo il cammino da u a v e da v a w.

In un grafo non orientato, le componenti connesse sono le classi di equivalenza della relazione di raggiungibilità. Una visita di un grafo restituisce esattamente una componente connessa di quel grafo (l'albero di visita).

Algoritmo 13 COMPONENTI CONNESSE(G)

```
INIZIALIZZA(G)
Π ← foresta vuota
for ogni u ∈ V do
    if color[u] = white then
        π_u ← VISITA(G,u) (π_u è l'albero di visita)
        Π ← Π + π_u
    end if
end for
return Π
```

Il grafo è connesso se e solo se nell'albero di visita tutti i vertici hanno un predecessore eccetto il nodo sorgente. La complessità è $\mathcal{O}(m+n)$.

Algoritmo 14 CONNESSIONE(G)

```
INIZIALIZZA(G)
scegli vertice s appartenente a G
π ← VISITA(G,s)
for ogni u appartenente a G do
    if u ≠ s e π[u]=NULL then return false
    end if
end for
return true
```

5.2 Componenti fortemente connesse

In un grafo orientato G, due nodi, u e v si dicono mutualmente raggiungibili o **fortemente connessi** se ognuno dei due è raggiungibile dall'altro, cioè esiste un cammino da u a v e da v a u ($u \leftrightarrow v$). La connessione forte è una relazione di equivalenza. Una **cfc** di un grafo orientato G è un sottografo G' di G fortemente connesso e massimale: i nodi di G' sono tutti fra loro fortemente connessi e nessun altro nodo di G è fortemente connesso con nodi di G'. Ogni vertice del grafo fa parte di una cfc, in quanto ogni nodo è almeno fortemente connesso con se stesso. Da un grafo fortemente connesso è possibile ricavare un DAG (e quindi un ordinamento topologico).

Per trovare la cfc contenente un vertice x:

- calcoliamo i **discendenti** di x ($\mathrm{D}(x)$), ossia i vertici di G raggiungibili da x
- calcoliamo gli **antenati** di x ($\mathrm{A}(x)$), ossia i vertici che raggiungono x
- cfc[x] è data dall'**intersezione** tra l'insieme degli antenati e dei discendenti ($D(x) \cap A(x)$)

Questo algoritmo è molto costoso, $\mathcal{O}(n^2+m)$.

Lemma (Cammino fortemente connesso)**.** *Se due vertici x, y di un grafo sono in una stessa cfc, allora nessun cammino tra di essi può abbandonare tale cfc.*

Dimostrazione. Sia z tale che $x \to z$ e $z \to y$. z è banalmente raggiungibile da x per ipotesi ($x \to z$); siccome x, y appartengono alla stessa cfc, esisterà un cammino $y \to x$. Esiste anche $z \to y$ per ipotesi. Quindi, per concatenazione, esisterà anche un cammino $z \to y \to x$. □

Teorema (Sottoalbero fortemente connesso)**.** *In una qualunque DFS di un grafo G orientato, tutti i vertici di una cfc vengono collocati in uno stesso sottoalbero.*

Dimostrazione. Sia r il primo vertice di una data cfc che viene scoperto dalla DFS: da r sono raggiungibili tutti gli altri vertici della cfc (per definizione). Poichè r è il primo vertice ad essere stato scoperto, al momento della sua scoperta tutti gli altri vertici della cfc saranno bianchi. Per il lemma precedente, tutti i cammini da r agli altri vertici della cfc conterranno solo vertici bianchi che fanno parte della cfc. Allora, per il **teorema del cammino bianco**, tutti i vertici appartenenti alla cfc di r saranno discendenti di r nell'albero DFS. □

Proprietà (1)**.** *Esiste sempre, per ogni grafo diretto, almeno un ordine di visita DFS dei suoi nodi tale per cui le cfc sono già separate nella foresta di visita.*

Proprietà (2)**.** *Un grafo G ed il suo trasposto G^T hanno le stesse cfc.*

Siano x e y due vertici di un grafo G; assumiamo che x non sia sulla stessa cfc di y. Dopo la DFS su G si possono presentare i seguenti casi:

1. y **è discendente di** x in un albero della foresta DFS di G. Esiste un cammino da x a y, ma non il contrario, quindi non esisterà un cammino da x a y in G^T ($d[x] < d[y] < f[y] < f[x]$).
2. x **e** y **non sono uno discendente dell'altro** nella foresta DFS di G. Non può esistere nessun cammino da y a x, altrimenti x sarebbe nel sottoalbero di y. Quindi, non eeisterà il cammino da x a y in G^T ($d[y] < f[y] < d[x] < f[x]$).

In entrambi i casi, ($f[x] > f[y]$), quindi nella seconda visita i vertici saranno considerati in ordine decrescente di tempo di fine visita.

Da queste osservazioni, si ricava l'**algoritmo di Kosaraju**:

1. visita G con l'algoritmo VISITA TUTTI I VERTICI DFS e costruisci una lista di vertici in ordine decrescente dei tempi di fine visita
2. costruisci G^T
3. visita G^T con l'algoritmo VISITA TUTTI I VERTICI DFS, considerando i vertici nell'ordine trovato al passo 1

La complessità è pari a $\mathcal{O}(n+m)$. Per dimostrare la correttezza dell'algoritmo, ci si avvale del **teorema del sottoalbero fortemente connesso** e dei lemmi seguenti.

Lemma (2)**.** *Un grafo orientato e il suo trasposto hanno le stesse cfc.*

Lemma (3)**.** *Sia A^T un albero ottenuto con la visita in profondità G^T, considerando i vertici in ordine decrescente dei tempi di fine visita su G, e sia u la sua radice. Per ogni vertice v discendente di u in A^T, v e u appartengono alla stessa cfc.*

Dimostrazione. Dimostriamo che ogni discendente di u in A^T è anche un discendente di u in un albero della foresta costruita dalla DFS su G. La dimostrazione è fatta per assurdo.
Consideriamo un cammino sull'albero A^T a aprtire dalla radice u; sia v il primo vertice sul cammino per cui il lemma non vale (cioè v non è discendente di u nella visita di G) e sia w il suo predecessore sul cammino. L'enunciato è valido per w: $d[u] \leq d[w] < f[w] \leq f[u]$.
Siccome la visita DFS di G^T considera i vertici in ordine decrescente di fine visita, vale $f[v] < f[u]$. Per il **teorema delle parentesi**, se v non è discendente di u nella prima visita, deve valere $d[v] < f[v] < d[u] < f[u]$. Ma questo è impossibile, in quanto v è adiacente a w in G, e la visita di v non può terminare prima che sia iniziata la visita di un suo adiacente.
Quindi in G esiste un cammino da u a v. Siccome v è discendente di u in A^T, esiste anche un cammino da u a v in G^T, e quindi da v a u in G. □

L'algoritmo è quindi corretto.

6 Tecnica greedy

6.1 Problemi

Un **problema P** è definito come una relazione $P \subseteq I \times S$: I è l'insieme delle possibili **istanze** del problema e l'insieme S è l'insieme delle possibili **soluzioni** del problema. Tipi di problemi:

- **problemi di decisione**: problemi che richiedono di verificare una certa proprietà sull'input (risultato o vero o falso)
- **problemi di ricerca**: data un'istanza del problema, restituire una soluzione ammissibile
- **problemi di ottimizzazione**: data un'istanza di un problema ed una funzione obiettivo valuta(Sol), che per ogni soluzione Sol restituisce un valore di tale soluzione Sol ottima

Una soluzione di un problema di ottimizzazione è anche una soluzione di un equivalente problema di ricerca (ma non viceversa).

6.2 Problema dello zaino frazionario

6.2.1 Definizione

Un ladro entra in un magazzino e trova n oggetti. L'i-esimo oggetto o_i ha un valore di c_i euro e pesa p_i chilogrammi. $v_i = c_i/p_i$ è il valore per unità di peso. Gli oggetti sono frazionabili, quindi il ladro ne può prendere anche solo una frazione x_1, $0 \leq x_i \leq 1$. In tal caso, il valore della parte presa sarà $c_i x_i$. Il ladro ha un solo zaino, che può contenere oggetti per un peso massimo di P chilogrammi.

Quali oggetti e in quale quantità dovrà prendere il ladro per ottenere il massimo guadagno dal furto?

6.2.2 Formalizzazione

Il problema richiede di valorizzare x_i per ogni $1 \leq i \leq n$ in modo che $\sum x_i c_i$ sia massima. Due proprietà:

- **Ottimalità**. Tutte le soluzioni che massimizzano il valore totale dello zaino (la **funzione obiettivo**) sono soluzioni ottime.
- **Ammissibilità**. Tutte le valorizzazioni dei vari x_i tali che il peso sia minore di P e che x_i sia compreso tra 0 e 1 sono soluzioni ammissibili.

6.3 Sottostruttura ottima e scelta greedy

Un problema ha la proprietà della **sottostruttura ottima** se una soluzione ottima del problema include le soluzioni ottime dei suoi sottoproblemi.
Se un problema di ottimizzazione gode delle proprietà della sottostruttura ottima, in genere può essere risolto con tecniche di tipo **greedy** i di **programmazione dinamica**.

Un problema gode della proprietà della **scelta greedy** se è sempre possibile scegliere in esso una variabile decisionale, attribuirle un valore ammissibile e localmente ottimo, toglierla dal problema ed ottenere un sottoproblema la cui soluzione, unita al valore di questa variabile, sia una soluzione ottima per il problema originario.

Viene definito un **criterio di appetibilità**, che permette di effettuare efficientemente la scelta greedy. L'appetibilità ordina le variabili in maniera che in ogni sottoproblema sia possibile effettuare efficientemente la scelta greedy.

6.4 Schema di algoritmo greedy

Passi di un algoritmo greedy:

1. scegli la variabile del problema con appetibilità maggiore
2. attribuisci ad essa il valore ammissibile più promettente a livello locale
3. ottieni il sottoproblema risultante dall'eliminazione della variabile al punto 1
4. se il problema è risolto, restituisci i valori delle variabili assegnati al punto 2, altrimenti vai al punto 1

Nella maggior parte dei casi, gli algoritmi greedy affrontano problemi dove, dato un insieme di elementi A, è necessario selezionare un sottoinsieme S di elementi ammissibile e ottimo. Le appetibilità possono

Algoritmo 15 GREEDY(A)

```
S ← insieme vuoto
while S non è una soluzione do
    estrai da A l'elemento più appetibile a
    if S ∪ a è ammissibile then
        aggiungi a all'insieme S
    end if
end while
```

essere fisse o modificabili.

6.5 Massimo numero di intervalli disgiunti

6.5.1 Problema

Dato un insieme di intervalli, trovare il massimo numero di intervalli disgiunti significa trovare un sottoinsieme costituito da intervalli tutti disgiunti, tale che il numero di intervalli sia massimo.

La soluzione ottimale si ottiene scandendo l'insieme di intervalli scegliendo ogni volta l'intervallo che finisce prima:

1. orina l'insieme degli intervalli in una sequenza S ordinata secondo l'istante finale
2. inizializza la soluzione Sol come sequenza vuota
3. scandisci S in ordine, e per ogni suo elemento A:
   (a) se A inizia dopo la fine dell'ultimo elemento di Sol, aggiungilo in fondo a Sol
   (b) altrimenti, non aggiungerlo

La complessità è quella dell'ordinamento del passo 1 più quella della scansione di S al passo 3, $\mathcal{O}(n \log n)$.

6.5.2 Correttezza

Definiamo le invarianti (S è l'insieme di intervalli esaminati ad un passo intermedio k):

- **Max**: la sequenza $A_1, A_2, \ldots, A_k$ di intervalli disgiunti scelti è, per l'insieme S, una **sequenza massimale**
- **PrimaMax**: la sequenza $A_1, A_2, \ldots, A_k$ è, fra le sequenze massimali, quella che finisce prima

- **PrimaVisti**: ogni intervallo $\in S$ termina prima della fine di qualunque intervallo $\notin S$

L'invariante da dimostrare è **Max**.

Dimostrazione. **Caso base**. Inizialmente, S è l'insieme vuoto, la sequenza massimale per S è la sequenza vuota. Tutte e tre le invarianti sono vere.
Passo. Sia A l'intervallo che termina prima tra quelli ancora da esaminare (ossia fra quelli $\notin S$). Consideriamo allora l'insieme $S' \equiv S \cup \{A\}$ e cerchiamo di stabilire qual è per S' la sequenza massimale (**Max**) che finisce prima (**PrimaMax**).
Caso 1: A inizia **prima** della fine della sequenza $A_1, A_2, \ldots, A_k$, quindi A interseca A_k; quindi, ogni sequenza avente A come ultimo elemento non può avere più di k elementi. Non inserendo A nella soluzione, l'invariante Max si mantiene.
Caso 2: A inizia **dopo** la fine della sequenza $A_1, A_2, \ldots, A_k$; allora, la sequenza $A_1, A_2, \ldots, A_k, A$:

- è per S' una sequenza massimale, perchè una sequenza massimale per S' non può avere più di $k+1$ elementi
- è per S' la sequenza massimale che finisce prima, perchè in S' non ci sono sequenze massimali non includenti A

Anche in questo caso, le invarianti si mantengono. □

6.6 Algoritmo di Moore

6.6.1 Problema dello scheduling e algoritmo risolvente

Una **sequenza di shceduling**, o semplicemente **scheduling**, è una sequenza di lavori $L_1, L_2, \ldots, L_n$ che devono essere eseguiti uno dopo l'altro consecutivamente. L'algoritmo è greedy, l'appetibilità è data dalla scadenza. Descrizione informale:

1. ordina la sequenza dei lavori per ordine crescente di istante di scadenza
2. scandisci tale sequenza in ordine, aggiungendo ogni volta il successivo lavoro alla fine della sequenza di scheduling provvisoria; se così facendo il lavoro considerato termina dopo la scadenza, si elimina dalla sequenza di scheduling il lavoro di durata massima

Algoritmo 16 MOORE()

```
Sol ← sequenza vuota
for i=n to n do
    aggiungi L_i a Sol
    t = t+durata di L_i
    if t >scadenza di L_i then
        togli da Sol il lavoro L_max di durata massima
        t = t−durata di L_max
    end if
end for
```

6.6.2 Correttezza

L'algoritmo di Moore è corretto, si dimostra per induzione.

Dimostrazione. **Invariante**. Sia S l'insieme di tutti i job finora esaminati:

1. *Sol* è uno **scheduling massimale** $L_1, L_2, \ldots, L_k$ di job di S che rispetta le scadenze, cioè quello con il numero massimo di elementi

2. *Sol* è, fra tutti gli scheduling massimali di job di S, quello di durata totale minima

3. *Sol* è ordinato per tempi di scadenza crescenti

4. ogni job non in S ha una scadenza posteriore o uguale alle scadenze dei job in S

Passo. Sia t_k l'istante di fine dello scheduling $Sol = L_1, L_2, \ldots, L_k$. Sia L il primo job non ancora esaminato, cioè non in S, che ha scadenza prima di tutti gli altri job non esaminati. Siano s la scadenza di L e d la durata di L. Considerando l'insieme $S' = S \cup \{L\}$, si possono verificare due casi:

1. L aggiunto ad S è eseguibile entro la sua scadenza

2. L non è eseguibile

Caso 1. Lo scheduling così ottenuto è uno scheduling massimale. $\{L\}$ ha cardinalità 1; quindi, se esistesse uno scheduling per $S \cup \{L\}$ con più di $k+1$ elementi, vi sarebbe uno scheduling per S con più di k elementi, il che è assurdo. Inoltre $S \cup \{L\}$ è di durata minima.
Caso 2. Lo scheduling $L_1, L_2, \ldots, L_k, L$ non è una soluzione. Se L è il processo di durata massima, sostituendolo ad un job già presente S, si otterrebbe uno scheduling di durata maggiore o uguale (violando punto 2). Se in S esiste un job L_{max} di durata maggiore di L, eliminando L_{max} ed aggiungendo L ad S, si ottiene uno scheduling di ancora k elementi, di durata minore (minima).
Quindi, $L_1, L_2, \ldots, L_k$ è uno scheduling massimale e di durata minima. □

6.7 Codici di Huffman

6.7.1 Codifica

Una **codifica** associa un insieme di caratteri o simboli ad un insieme di altri elementi, detti **parole in codice**. Le codifiche possono avere lunghezza fissa o variabile. La codifica deve essere non ambigua, cioè che nessuna codifica di di un carattere sia prefisso di un'altra. Un codice binario prefisso può essere rappresentato tramite un albero binario in cui le foglie rappresentano i caratteri ed i cammini dalla radice alle foglie rappresentano la codifica dei caratteri. L'albero prende il nome di **albero di codifica**.

6.7.2 Codifica ottima

Il problema affrontato dall'algoritmo di Huffman è il seguente: dato un testo scritto secondo un certo alfabeto C, trovare una codifica che sia minimale, cioè renda minima la lunghezza del testo codificato (è una tecnica di compressione).

Una codifica a lunghezza fissa usa parole in codice tutte della stessa dimensione. Con questa codifica servono almeno $\lceil \log_2 n \rceil$ bit per rappresentare ogni parola in codice per un alfabeto di n elementi.

Un albero avente nodi interni con un solo figlio non è ottimale: può essere eliminato attaccando i suoi figli direttamente al padre. Un albero binario in cui ogni nodo interno ha esattamente due figli si chiama **albero pieno**.

Per ottenere la maggior compressione possibile, i caratteri più frequenti devono avere le codifiche più corte, cioè comparire a livelli più alti dell'albero.

6.7.3 Algoritmo di Huffman

L'algoritmo di Huffman ha come input: un alfabeto, cioè un insieme C di caratteri; una funzione f che da' la frequenza di ciascun carattere in un dato testo t. Produce in output un codice binario ottimo per

la compressione di quel testo t. Con $f(c)$ = frequenza con cui il carattere c compare nel testo e d_c = livello del carattere c nell'albero T, si chiama lunghezza media di codifica o costo di T:

$$*L(T) = \Sigma_{c \in C} d_C \cdot f(c) \tag{1}$$

Se n è il numero di caratteri che compongono il testo con frequenze date dalla funzione f, la lunghezza in bit della codifica del testo è data da:

$$*B(T) = \Sigma_{c \in C} d_C \cdot n \cdot f(c) = n \cdot L(T) \tag{2}$$

L'algoritmo di Huffman è un'applicazione della tecnica greedy **con appetibilità modificabili**:

1. Per ciascun carattere, crea un albero formato solo da una foglia contenente il carattere e la frequenza del carattere
2. Fondi i due alberi che hanno due frequenze minime e costruisci un nuovo albero che ha come frequenza la somma delle frequenze degli alberi fusi
3. Ripeti la fusione finchè si ottiene un unico albero

Algoritmo 17 HUFFMAN(C,f)

```
n ← |C| (insieme di caratteri)
Q ← coda di priorità (heap) vuota
for ogni carattere in C do
    enqueue(Q,createTreeNode(c,NULL,NULL),f[c])
end for
for i = 0; i < n − 1; i++ do
    x ← dequeueMin(Q)
    y ← dequeueMin(Q)
    z ← createTreeNode(null,x,y)
    f[z] ← f[x] + f[y]
    enqueue(Q,z,f[z])
end for
return dequeueMin(Q)
```

La complessità dell'algoritmo è $\mathcal{O}(\log n)$ se la coda di priorità è realizzata con un heap.

6.7.4 Correttezza

L'algoritmo restituisce un albero di Huffman, che rende minima la lunghezza media di codifica.

Foresta di Huffman. La foresta di Huffman per un alfabeto C con funzione di frequenza f è una foresta i cui elementi $T_1, T_2, \ldots, T_n$ sono sottoalberi di un albero di Huffman T per quell'alfabeto.

Invariante. La foresta $\{T_1, T_2, \ldots, T_n\}$ costruita dall'algoritmo al passo generico è una foresta di Huffman per C e f, cioè esiste un albero di Huffman T di cui gli alberi $T_1, T_2, \ldots, T_n$ sono sottoalberi.

Caso base. Prima dell'esecuzione del ciclo, l'invariante vale banalmente, in quanto gli alberi sono tutti nodi singoli, cioè foglie corrispondenti ai caratteri dell'alfabeto C.

Passo. Assumiamo che prima della k-esima iterazione del ciclo l'invariante valga, cioè che la foresta $F = \{T_1, T_2, \ldots, T_n\}$ sia una foresta di Huffman per il dato alfabeto C. Mostriamo che dopo il (k+1)-esimo passo dell'iterazione, che fonde i due alberi T_a e T_b aventi due frequenze minime in un nuovo albero T_{ab}, la nuova foresta $F - \{T_a, T_b\} \cup \{T_{ab}\}$ è ancora una foresta di Huffman.

Dimostrazione. Mostriamo che fra tutti gli alberi di codifica di cui $T_1, T_2, \ldots, T_n$ sono sottoalberi, vi è un albero (T''') la cui lunghezza media di codifica $L(T''')$ è minima e di cui T_{ab} è un sottoalbero. Quindi l'albero T_{ab} può essere inserito nella foresta al posto di T_a e T_b: la foresta risultante $F - \{T_a, T_b\}$ è ancora una foresta di Huffman.
Per ipotesi induttiva, esiste un albero T' di cui gli alberi $T_1, \ldots, T_a, \ldots, T_b \ldots, T_n$ sono sottoalberi, dove T_a e T_b sono, nella foresta al passo considerato, i due alberi di frequenze minime, $f(T_a)$ e $f(T_b)$. Mostriamo che esiste un albero di Huffman T''' avente T_{ab} come sottoalbero.
Consideriamo, fra i nodi interni di T' non appartenenti alla foresta F, cioè fra i nodi che nel passo considerato non sono ancora stati creati, quello di profondità massima. Sia esso z. Come ogni nodo interno, z deve avere due sottoalberi figli non nulli, siano T_x e T_y.
Poichè T_a e T_b sono, al passo considerato, i due alberi di pesi minimi, assumendo $f(T_a) <= f(T_b)$ e $f(T_x) <= f(T_y)$, abbiamo $f(T_a) <= f(T_x)$ e $f(T_b) <= f(T_y)$. Poichè z è un nodo di profondità massima, le radici degli alberi T_x e T_y si trovano in T' a profondità d non inferiore a quelle di T_a e T_b, siano d_1 e d_2: quindi, $d_1 <= d$ e $d_2 <= d$.
Scambiamo di posizione T_a e T_x per ottenere un albero di lunghezza media non superiore (analogamente per T_b e T_y).
Ma T' è un albero di Huffman, cioè avente $L(T')$ minimo. Quindi deve essere $L(T''') = L(T')$, e anche T''', che ha T_{ab} come sottoalbero, è un albero di Huffman. Dunque, la foresta $F - \{T_a, T_b\} \cup \{T_{ab}\}$ è ancora una foresta di Huffman. □

7 Tecnica Greedy applicata ai grafi

Il teorema della distanza nell'albero BFS visto per i grafi non pesati non vale per un grafo pesato, quindi $d[v] \neq \delta(s, v)$.

7.1 Algoritmo di Dijkstra

L'algoritmo di Dijkstra è una visita BFS in cui la frangia è gestita come in un algoritmo greedy con appetibilità modificabili. L'appetibilità di un vertice u è data da una stima della distanza tra s e u che l'algoritmo ha in un determinato istante. Inizialmente, tutti i vertici sono stimati a distanza ∞ da s, tranne s stesso che ha distanza $d[s] = 0$ da se stesso. Ad ogni ciclo, scelgo un vertice u da aggiungere all'albero tra quelli non ancora inseriti ma aggiungi dalla ricerca, e scelgo quello con **distanza da** s **stimata minima**.

Algoritmo 18 DIJKSTRA(G,W,s)

```
INIZIALIZZA(G)
color[s] ← gray
d[s] ← 0
while esistono vertici grigi do
    u ← vertice grigio con d[u] minore
    S ← S ∪ {u}
    for ogni v adiacente ad u do
        if color[v] ≠black then
            color[v] ← gray
            if d[v] > d[u] + W(u, v) then
                π[v] ← u
                d[v] ← d[u] + W(u, v)
            end if
        end if
    end for
```

$color[u] \leftarrow$ black
end while

7.1.1 Miglioramenti

I nodi grigi sono gestiti con una **coda di priorità**. È possibile non distiguere nodi bianchi e grigi ed inserire tutti nella coda fin dall'inizio, assegnando loro distanza infinita da s: i nodi neri sono detti **definitivi** e quelli bianchi e grigi **non definitivi**. Inoltre, non è nemmeno necessario distinguere i nodi neri, poichè non saranno in coda.

Algoritmo 19 DIJKSTRA CON PRIORITY QUEUE(G,W,s)

```
INIZIALIZZA(G)
D ← coda di priorità vuota
d[s] ← 0
for ogni v in v[G] do
    enqueue(D,v,d[v])
end for
while NotEmpty(D) do
    u ← dequeueMin(D)
    S ← S ∪ {u}
    for ogni v adiacente ad u do
        if d[v] > d[u] + W(u,v) then
            π[v] ← u
            d[v] ← d[u] + W(u,v)
            decreaseKey(D,v,d[v])
        end if
    end for
end while
```

7.1.2 Complessità

Denotiamo con:

- t_c: tempo impiegato dalla costruzione della coda
- t_e: tempo impiegato da una estrazione del minimo
- t_d: tempo impiegato da una decreaseKey

Ad ogni ciclo della visita bisogna estrarre il minimo dalla coda; per ogni arco trovato potrebbe essere necessario decrementare la chiave di un vertice. La complessità è quindi: $\mathcal{O}(t_c + n * t_e + m * t_d)$.

La complessità totale dipende anche dalle complessità delle operazioni sulla coda:

- con coda di priorità realizzata come sequenza **non ordinata**: $\mathcal{O}(n^2 + m)$
- con coda di priorità realizzata come sequenza **ordinata**: $\mathcal{O}(n + n * m)$

7.1.3 Versione di Johnson

La versione dell'algoritmo di Dijkstra con coda di priorità implementata come **heap** è chiamata algoritmo di Johnson. La complessità è $\mathcal{O}((m + n) \log n)$.

7.1.4 Correttezza

Proprietà (1)**.** *Un sottocammino di un cammino minimo è un* ***cammino minimo****.*

Proprietà (2)**.** *Siano S l'insieme di vertici già considerati dalla visita e D l'insieme dei vertici ancora da considerare:*

2.1. il $d[v]$ di ogni vertice in S non viene più modificato

2.2. tutti i predecessori dei nodi nella coda di priorità sono in S

2.3. per ogni nodo (eccetto s), $d[v] \neq \infty$ se e solo se il predecessore di quel nodo non è nullo

2.4. per ogni nodo (eccetto s), $d[v] \neq \infty$ implica $d[v] = d[\pi[v]] + W(\pi[v], v)$

Proprietà (3)**.** *Se un cammino ha distanza diversa da ∞, esiste un cammino tra due nodi nel grafo.*

Dimostrazione. Per ipotesi induttiva, esiste un cammino da s a $\pi[u]$. Allora, il cammino da s a $\pi[u]$ più l'arco $(\pi[u], u)$ costituisce un cammino da s a u.
Supponiamo, per assurdo, che tra s e u vi sia almeno un cammino $s \equiv v_1 \to v_2 \to \cdots \to v_{k-1} \to v_k \equiv u$ e che u venga estratto dalla coda con $d[u] = \infty$. v_{k-1} è già stato estratto quindi $d[u] = d[v_k] = d[v_{k-1}] + W(v_{k-1}, v_k)$. Ma $W(v_{k-1}, v - k) < \infty$ e $d[u] = \infty$, quindi $d[v_{k-1}] = \infty$. Questo può essere iterato per ciascun vertice del cammino, contraddicendo $d[s] = 0$. □

Il predicato $\forall t \in S : d[t] = \delta(s, t)$ è un'invariante del ciclo while.

Dimostrazione. **Caso base**. Il predicato è vero perchè all'inizion S è vuoto.

Passo. Dimostriamo che per il nuovo vertice u estratto da D, $d[u] = \delta(s, u)$.

Caso 1. Il vertice estratto da D ha $d[u] \neq \infty$. Sia $\pi[u] = r \neq NULL$ per proprietà 2.3. Sappiamo allora che r è nell'albero dei cammini minimi e che $d[u] = d[r] + W(r, u)$ (per proprietà 2.4).
Supponiamo, per assurdo, che tra s e u esista un cammino di peso minore di $d[u]$: esso deve contenere un arco che tra un vertice in s e uno in D, poniamo siano x il vertice in S e y quello in D. Questo cammino può essere visto come la concatenazione di tre cammini $(s \rightsquigarrow x \rightsquigarrow y \rightsquigarrow u)$. Se $s \rightsquigarrow x \rightsquigarrow y \rightsquigarrow u$ è minimo, anche $s \rightsquigarrow x \rightsquigarrow y$ è minimo, quindi $d[y] = \delta(s, y)$. Si ottiene quindi $d[y] + W(y \rightsquigarrow u) \geq d[y] \geq d[u]$, perchè u è stato estratto e ha $d[u]$ minimo.
Quindi $W(s, \ldots x, \ldots y, \ldots u) = d[y] + W(y \rightsquigarrow u)$ non può essere minore di $d[u]$.

Caso 2. Se u è estratto con $d[u] = \infty$, allora (proprietà 3) non esiste alcun cammino tra s e u. □

7.1.5 Ulteriori ottimizzazioni

L'algoritmo si può ottimizzare nel caso serva solamente trovare un cammino minimo tra due nodi, s e t:

- Ogni volta che si estrae un nodo u da D, si può controllare se **coincide con** t e terminare l'algoritmo
- Poichè l'algoritmo esegue una BFS, si vanno a calcolare prima tutti i cammini di lunghezza inferiore a $\delta(s, u)$: si possono usare le **euristiche** per individuare i nodi più promettenti.

Dati s e t, si può eseguire l'algoritmo contemporaneamente sul grafo G a partire da s e sul grafo trasposto G^T a partire da t, alternando i passi. Quando si ottengono un cammino minimo da s ed uno da t ad uno stesso nodo v, questo cammino è un cammino da s a t in G.

Teniamo in memoria la lunghezza del migior cammino così trovato, inizialmente $d = \infty$. Ogni volta che (in avanti) si percorre un arco (u, v) con nodo u nero (in avanti) e nodo v nero (in indietro), se $d[u] + W(u, v) + d^T[v] < d$ si aggiorna d. Quando si estraggono un nodo x (in avanti) e un nodo y (in indietro) tali che $\delta(s, x) + \delta(y, t) > d$, allora il cammino trovato di peso d è il cammino minimo.

7.2 Minimo albero ricoprente

7.2.1 Definizione

Dato un grafo G non orientato e connesso, un albero ricoprente è un **sottografo** $T \subseteq G$ tale che T è un sottografo connesso aciclico e T contiene tutti i vertici di G.

Dato un grafo G connesso, non orientato e pesato, il **minimo albero ricoprente** (MAR) per G è un albero ricoprente in cui la somma dei pesi degli archi nell'albero è minima.

7.2.2 Lemma del taglio

Un **taglio** è una partizione dell'insieme V di tutti i nodi del grafo in due parti non vuote, S e $V - S$. Si dice che un arco (u, v) **attraversa il taglio** se i suoi estremi u e v appartengono uno ad una parte ed uno all'altra.

Sia A un insieme di archi appartenenti ad un MAR di un grafo G. Consideriamo un taglio non attraversato da alcun arco di A; siano S e $V - S$ le sue due parti. Sia (u, v) l'arco di peso minimo fra tutti gli archi del grafo che attraversano il taglio (chiamato **arco leggero**): allora (u, v) appartiene a un MAR che estende A, cioè l'insieme $A \cup \{(u, v)\}$ è anch'esso un sottoinsieme di un MAR del grafo G.

Proprietà (1)**.** *Se in un albero libero si elimina un arco, si ottengono due alberi.*

Proprietà (2)**.** *Se si connettono due alberi, si ottiene un albero.*

Dimostrazione. Dato un grafo G, siano:

- A: insieme di archi di G che supponiamo siano appartenenti ad uno stesso MAR di G
- $(S, V - S)$: un taglio che non taglia nessun arco di A (ma taglia archi di G)
- (u, v): arco di peso minimo fra quelli di G tagliati dal taglio $(S, V - S)$

Un MAR di G che estende A è per definizione un albero di peso minimo fra tutti gli alberi ricoprenti di G che contengono A. Un albero ricoprente AR contenente A deve connettere tutti i nodi di G, quindi deve contenere un cammino tra u e v. Poichè u e v si trovano da parti opposte del taglio, un tale cammino deve contenere almeno un arco (x, y) che attraversa il taglio (possono anche coincidere). Se nell'albero AR sostituiamo l'arco (x, y) con l'arco (u, v), si ottiene ancora un albero ricoprente di peso totale minore o uguale al peso di AR. □

Poichè l'arco (u, v) non appartiene ad A, l'insieme $A \cup \{(u, v)\}$ contiene un arco in più rispetto ad A. Aggiungendo un arco alla volta si può quindi costruire un MAR ricoprente G.

Algoritmo 20 TROVA MAR(G)

```
A ← insieme di archi vuoto
while A non contiene tutti i vertici di G do
    trova un taglio non attraversato da alcun arco di A
    aggiungi ad A un arco di peso minimo fra quelli che attraversano il taglio
end while
```

Teorema (Unicità del MAR)**.** *Se i pesi degli archi sono tutti distinti, il MAR è unico.*

Dimostrazione. Per assurdo, supponiamo che G abbia due minimi alberi ricoprenti distinti, $M1$ e $M2$. Poichè sono distinti, esiste almeno un arco in uno dei due alberi che non appartiene anche all'altro. Sia e l'arco di peso minimo che appartiene solo ad uno dei due MAR (in questo caso, $M1$). Se si aggiunge e ad $M2$, si crea un ciclo C. Poichè $M1$ non contiene cicli, nel ciclo c'è almeno un arco e' che non appartiene

a $M1$: questo arco ha peso maggiore di e in quanto abbiamo scelto e come arco di peso minimo che appartiene as uno dei due alberi ma non all'altro.
Togliendo e' dal ciclo, si ottiene un albero $M'2$ diverso da $M2$; $M'2$ ha un peso minore di $M2$, fatto che contraddice l'ipotesi iniziale. □

7.3 Algoritmo di Prim

Nei MAR si cerca non il nodo più vicino alla radice, ma il nodo più vicino all'albero già costruito; l'appetibilità è rappresentata dalla stima della distanza del nodo dall'albero di visita.

Non è possibile eliminare la gestione dei nodi neri in quanto bisogna controllare che tra gli adiacenti del nodo u estratto dalla coda non vi sia il genitore di u.

7.3.1 Confronto con l'algoritmo di Dijkstra

Nell'algoritmo di Prim, a differenza di quello di Dijkstra, $d[v]$ è il peso dell'arco minimo tra v e l'albero di visita già costruito nel momento in cui v viene estratto, ma questo non impedisce che in un secondo momento possa esistere un arco da un vertice chiuso a v di peso minore di $d[v]$. L'aggiornamento dell'appetibilità è simile a quello di Dijkstra, ma $d[v] \leftarrow W(u, v)$.

Algoritmo 21 PRIM(G,W,s)

```
INIZIALIZZA(G)
D ← coda di priorità vuota
d[s] ← 0
for ogni v in V[G] do
    enqueue(D,v,d[v])
    def[v] ← false
end for
while NotEmpty(D) do
    u ← dequeueMin
    S ← S ∪ {u}
    def[v] ← true
    for ogni v adiacente ad u do
        if def[v] =false and d[v] > W(u,v) then
            π[v] ← u
            d[v] ← W(u,v)
            decreaseKey(D,v,d[v])
        end if
    end for
end while
```

7.3.2 Complessità

La complessità dell'algoritmo di Prim è la stessa di quella dell'algoritmo di Johnson, quindi $\mathcal{O}((m + n) \log n)$. Siccome il grafo è connesso, $m \geq n - 1$, quindi $\mathcal{O}(m \log n)$.

7.3.3 Correttezza

Sia S l'albero di visita costruito. S conterrà i nodi definitivi (quelli nell'albero) e $V - S$ contiene i nodi non definitivi (non nell'albero).

Invarianti:

- **IS**: tutti gli archi dell'albero S appartengono ad un qualche minimo albero ricoprente dell'intero grafo G
- **ID**: per ogni nodo x non definitivo, $d[x]$ è il peso dell'arco più leggero che collega x ad un nodo nero

Dimostrazione. **Caso base**. Dopo la prima iterazione, c'è solo un nodo definitivo, s; l'albero S non contiene nessun arco; ogni nodo x adiacente a s ha distanza $d[x]$ uguale al peso dell'arco (s, x) e ogni altro nodo ha distanza uguale a ∞. Quindi, **IS** e **ID** sono soddisfatti.

Passo. Scegliamo un taglio che separi i nodi neri dagli altri: sicuramente non taglia nessun arco di S. Scegliamo u con $d[u]$ minore tra i nodi non neri. S è sottoinsieme di un MAR, e per il lemma del taglio (y, u) appartiene ad un MAR di G che estende S (IS si mantiene). Per ripristinare ID, basta controllare se il nuovo nodo u avvicina qualche suo adiacente ad S. □

7.4 Algoritmo di Kruskal

7.4.1 Introduzione

Considerando la definizione di MAR e la struttura generica di un algoritmo greedy, è possibile definire un algoritmo che costruisce iterativamente un MAR, a partire dalla foresta vuota alla quale si aggiungono via via degli archi con i seguenti criteri:

- **Appetibilità**: l'appetibilità di un arco è inversamente proporzionale al suo peso
- **Criterio di ammissibilità**: posso aggiungere un arco alla soluzione provvisoria solo se non crea cicli
- **Criterio per riconoscere la soluzione**: l'albero costruito è una soluzione solo se è connesso

7.4.2 Controllo dei cicli con UnionFind

L'algoritmo mantiene una foresta di alberi (A) che man mano vengono fusi dall'aggiunta di archi. Si crea un ciclo se viene aggiunto un arco tra due nodi che appartengono allo stesso albero. Si può usare la **UnionFind** per identificare gli insiemi di vertici che già appartengono allo stesso albero. Ogni qualvolta si vuole aggiungere un arco (u, v), di considera find(u) e find(v): se sono uguali, u e v appartengono allo stesso albero.

Algoritmo 22 KRUSKAL CON UNIONFIND(G)

```
A ← insieme di archi vuoto
oridina gli archi in una sequenza S in oridne non decrescente di peso
crea una UnionFind contenente i vertici di G come insiemi iniziali
for ongi arco (u, v) ∈ S do
    if find(u) ≠ find(v) then
        aggiungi (u, v) ad A
        union(u,v)
    end if
end for
```

7.4.3 Complessità

s

7.4.4 Correttezza

Invariante

Caso base

Passo

8 Programmazione dinamica

8.1 Tecnica della programmazione dinamica

8.1.1 Memoizzazione

La **memoizzazione** consiste nel memorizzare i valori via via calcolati da un algoritmo in una struttura di supporto. È possibile applicare la tecnica della memoizzazione a tutte quelle funzioni che non cambiano comportamento, sugli stessi input, nel tempo.

8.1.2 Massimo Sottoinsieme Indipendente

Si consideri un grafo G con struttura lineare. Ad ogni nodo v_i è associato un peso non negativo w_i. Dato un grafo con questa struttura, si determini il sottoinsieme indipendente di nodi del grafo (S) che ha massimo peso.

Questo problema gode della proprietà della sottostruttura ottima.

Teorema (Sottostruttura ottima MSI)**.** *Se S_i è una soluzione ottima per P_i (problema ristretto ai primi i nodi), allora vale una delle seguenti affermazioni:*

- *$V_i \notin S_i$ e S_i è anche una soluzione ottima per P_{i-1}*
- *$V_i \in S_i$ e $S_i - \{V_i\}$ è una soluzione attima per P_{i-2}*

Dimostrazione. Se S_i è una soluzione ottima per P_i, allora sono possibili due casi.

Caso 1. Nel primo caso, S_i è anche una soluzione ammissibile per il problema P_{i-1}, in quanto non contiene V_i. Se non fosse ammissibile per P_{i-1}, esisterebbe una soluzione S'_{i-1} con peso maggiore di S_i, ma tale soluzione sarebbe una soluzione anche per P_i e sarebbe migliore di S_i, contraddicendo l'ipotesi che S_i sia la soluzione ottima per P_i.

Caso 2. Nel secondo caso, V_i fa parte di S_i, quindi $S' = S_i - \{V_i\}$ è una soluzione ottima per il problema P_{i-2}. V_{i-1} non può appartenere a S_i per la condizione di indipendenza sulle soluzioni. Per il problema P_{i-2}, S' è una soluzione ottima: se non lo fosse, esisterebbe una soluzione S'' per P_{i-2} di peso maggiore di S'. Allora $S'' \cup \{V_i\}$ sarebbe una soluzione per P_i e avrebbe peso maggiore di S_i (assurdo). □

Corollario. *Se S_{i-1} è una soluzione ottima per P_{i-1} e S_{i-2} è una soluzione ottima per P_{i-2}, allora la soluzione per P_i è la soluzione di peso massimo tra S_{i-1} e $S_{i-2} \cup \{V_i\}$.*

$$S_0 = \emptyset$$

$$S_1 = w_1$$

$$S_i = \left\{ \begin{array}{ll} S_{i-1} & \text{se } W(S_{i-1}) > W(S_{i-1}) + w_i \\ S_{i-2} \cup V_i & \text{altrimenti} \end{array} \right\}$$

Si può calcolare S_i a partire dai sottoproblemi S_{i-1} e S_{i-2}. La proprietà di sottostruttura ottima è fondamentale per poter esprimere le funzioni in una forma che permetta la memoizzazione.

Algoritmo 23 MSI(n,A)

$A[0] \leftarrow \{\}$
$A[1] \leftarrow \{V_1\}$
for i=2...n **do**
    $A[i] \leftarrow$maxPeso$(A[i-1], A[i-2] + V_i)$
end for
return $A[n]$

8.2 Longest Common Subsequence

Data una sequenza $S : a_1, \ldots, a_m$, una **sottosequenza** di S è una qualsiasi sequenza ottenuta da un S togliendo alcuni elementi. La sottosequenza deve rispettare l'ordine degli elementi della sequenza originale.

8.2.1 Definizione del problema

Date due sequenze $S1$ e $S2$, trovare la più lunga sequenza $S3$ che è sottosequenza sia di $S1$ che di $S2$. Notatione: $S3 = lcs(S1, S2)$.

8.2.2 Sottostruttura ottima

Siano $S1 : a_1, \ldots, a_m$ e $S2 : b_1, \ldots, b_n$ e $S3 : c_1, \ldots, c_k$; sono possibili due casi.

Caso 1. Nel primo caso, $a_m = b_n$. a_m sarà contenuto in $S3$, e occuperà l'ultima posizione di $S3$; quindi $c_k = a_m = b_n$. Dato che gli ultimi elementi della sequenze iniziali già appartengono alla lcs, si possono non considerare nel confronto con il resto delle sequenze. Quindi $S3 = lcs(S1_{m-1}, S2_{n-1}) + \{a_m\}$.

Caso 2. Nel secondo caso, $a_m \neq b_n$. In questo caso, possiamo dure che a_m o b_n non saranno contenuti in $S3$. Quindi, $S3$ sarà la sequenza più lunga tra $lcs(S1 - \{a_m\}, S2)$ e $lcs(S1, S2 - \{b_n\})$.

8.2.3 Struttura per memoizzazione

Per memorizzare la soluzione di tutti i sottoproblemi, si usa una **matrice** (LCS) di $m + 1$ righe e $n + 1$ colonne. La casella $LCS[i, j]$ contiene la più lunga sottosequenza comune dei due segmenti iniziali di lunghezze rispettive i e j. La casella 0-esima contiene la più lunga sottosequenza dei prefissi vuoti.

8.2.4 Ottimizzazioni

Per ottimizzare l'operazione di controllo sulla lunghezza delle lcs memoizzate, si può definire una nuova matrice L che mantenga in $L[i, j]$ la lunghezza delle lcs memoizzate. Ogni volta, $LCS[i, j]$ è uguale a $LCS[i-1, j-1] + a[i-1]$ oppure $LCS[i-1, j]$ oppure $LCS[i, j-1]$: è possibile utilizzare semplicemente delle frecce per indicare come si sta costruendo $LCS[i, j]$, senza dover copiare ogni volta l'elemento precendente.

8.2.5 Complessità

Lc costruzione della matrice ha costo $\mathcal{O}(mn)$, così come il suo popolamento. La ricostruzione della soluzione ha costo $\mathcal{O}(m + n)$: in totale, $\mathcal{O}(mn)$.

8.3 Zaino 0-1

Simile al problema dello zaino frazionario, ma con oggetti non frazionabili.

Algoritmo 24 LCS(a,b,m,n)

```
LCS ← nuova matrice di dimensioni m + 1 * n + 1
L ← nuova matrice di dimensioni m + 1 * n + 1
for i = 0 ... m do
    LCS[i, 0] ← []
    L[i, 0] ← 0
end for
for i = 1 ... m do
    for j = 1 ... m do
        if a[i − 1] == b[j − 1] then
            LCS[i, j] ←↖
            L[i, j] ← L[i − 1, j − 1] + 1
        else if L[i − 1, j] > L[i, j − 1] then
            LCS[i, j] ←↑
            L[i, j] ← L[i − 1, j]
        else
            LCS[i, j] ←←
            L[i, j] ← L[i, j − 1]
        end if
    end for
end for
return LCS[m, n]
```

8.3.1 Costruzione di algoritmi di programmazione dinamica

Passi da seguire per cotruire un algoritmo di programmazione dinamica:

1. descrivere la **struttura dati** necessaria per la **memoizzazione**
2. definire i **casi base** e le loro soluzioni banali
3. scrivere l'algoritmo che inizializzi la struttura di memoizzazione seguendo i casi base e successivamente la popoli in maniera bottom-up; infine restituire la soluzione

8.3.2 Costruzione zaino 0-1

Equazione ricorsiva per l'algoritmo:

$$V(i,j) = \left\{ \begin{array}{ll} V(i-1,j) & \text{se } j < p_i \\ max(V(i-1,j), v(i-1, j-p_i) + v_i) & \text{altrimenti} \end{array} \right\}$$

$V(i,j)$ ha due parametri: i è l'ultimo oggetto considerato, j è la capienza. La struttura di memoizzazione è quindi una matrice; la soluzione sarà contenura in $V[n,P]$. La matrice avrà dimensioni $(n+1)*(P+1)$ per includere i casi base.

Casi base:

- $i = 0$: nessun oggetto considerato, $V[0,j]$ per ogni $0 \leq j \leq P$
- $j = 0$: capienza 0, $V[i,0] = 0$ per ogni $0 \leq i \leq n$

Per sapere quali oggetti appartengono alla soluzione, basta utilizzare una matrice ausiliaria K.

Algoritmo 25 ZAINO(n,P,v[],p[])

```
V[] ← nuova matrice (n+1) * (P+1)
K[] ← nuova matrice (n+1) * (P+1)
for i = 0...n do
    V[i,0] = 0
    K[i,0] = 0
end for
for j = 0...n do
    V[0,j] = 0
    K[0,j] = 0
end for
for i = 1...n do
    for j = 1...P do
        V[i,j] = V[i-1,j]
        K[i,j] = 0
        if V[i,j] < V[i-1,j-p[i]] + v[i] then
            V[i,j] < V[i-1,j-p[i]] + v[i]
            K[i,j] = 1
        end if
    end for
end for
return V[n,P]
```

9 UnionFind

9.1 Definizione del problema

Il problema è mantenere una collezione di insiemi disgiunti sulla quale siano possobili le seguenti operazioni:

- **union(A,B)**: fonde gli insiemi A e B in un unico insieme $A \cup B$
- **find(x)**: restituisce il nome dell'insieme contenente l'elemento x
- **makeSet(x)**: crea il nuovo insieme $\{x\}$, avente x come unico elemento

Gli insiemi rimangono sempre disgiunti. Per distinguere tra diversi insiemi, si può individuare un elemento rappresentante, cosicchè union(a,b) fonda i due insiemi rappresentati dagli elementi a e b.

9.2 Implementazione

2 tipi di approcci fondamentali:

- privilegiare esecuzione efficiente dell'operazione di find (**QuickFind**)
- privilegiare esecuzione efficiente dell'operazione di union (**QuickUnion**)

9.2.1 QuickFind

Gli approcci di tipo **QuickFind** utilizzano alberi con due livelli per realizzare la UnionFind. La radice di ogni albero contiene il rappresentante di un insieme, le foglie rappresentano gli elementi dell'insieme. Il rappresentante è contenuto sia nella radice sia in una foglia.

Operazioni su QuickFind:

- makeSet(x): ccrea un nuovo albero composto da una radice ed una foglia, ed in entrambi posiziona x. Costo: $\mathcal{O}(1)$.
- find(x): accede alla foglia corrispondente all'elemento x, risale di un livello incontrando la radice e restituisce l'elemento contenuto in essa. $\mathcal{O}(1)$.
- union(a,b): considera l'albero A contenente a e B contenente b, per ogni forglia di B, sostituisce il puntatore al padre con un puntatore alla radice di A e cancella la radice di B. $\mathcal{O}(n)$.

9.2.2 QuickUnion

Gli approcci di tipo **QuickUnion** utilizzano alberi con più di 2 livelli per rappresentare le UnionFind. La radice di ogni albero contiene il rappresentante di un insieme. I nodi rappresentano gli elementi dell'insieme. Il rappresentante è conenuto solamente nella radice.

Operazioni su QuickUnion:

- makeSet(x): crea un nuovo albero composto da un unico nodi contenente x. Costo: $\mathcal{O}(1)$.
- find(x): accede alla foglia corrispondente all'elemento x, risale fino alla radice dell'albero e restituisce l'elemento contenuto nella radice. Costo: $\mathcal{O}(n)$.
- union(a,b): considera l'albero A contenente a e B contenente b, rende la radice di B figlio della radice di A. Costo: $\mathcal{O}(1)$.

9.3 Complessità e costo ammortizzato

Senza bilanciamenti sull'albero, bisogna considerare il caso peggiore, $\mathcal{O}(n)$. Ciò che interessa è il costo totale delle operazioni, ossia il **costo ammortizzato**.

9.3.1 Metodo dei crediti

Il **metodo dei crediti** viene usato per determinare il costo ammotizzato di una sequenza di operazioni, senza andare nel dettaglio. 1 credito vale $\mathcal{O}(1)$ passi di esecuzione. Le funzioni meno costose depositano crediti sugli oggetti, ed i crediti possono essere prelevati dalle funzioni più costose. Il costo ammortizzato è dato dalla somma di tutti i costi diviso per il numero di operazioni.

Vediamo n makeSet e $n-1$ makeUnion analizzati con il metodo dei crediti:

- **Numero massimo di cambi padre**. Ad ogni makeUnion, se una foglia cambia padre, il nuovo insieme che la contiene sarà grande almeno il doppio di quello di partenza. Quindi la foglia dopo k cambi di padre apparterrà ad un insieme di 2^k elementi. Siccome il numero totale di elementi è n, $2^k \leq n$, quindi $k \leq \log_2 n$.
- **Deposito crediti**. Assegniamo ad ogni makeSet un costo aggiuntivo di $\log_2 n$ crediti. Con n makeSet, si accumulano $n \log_2 n$ crediti.
- **Costo union**. Ogni union consuma 1 credito per il costo dell'operazione d cambio del padre. Poichè i cambi di padre sono limitati a $\log_2 n$ per foglia, si consumano un totale di $n \log_2 n$ crediti.

Costo di m find, n makeSet e $n-1$ makeUnion: $\mathcal{O}(m + n \log_2 n)$.

9.4 Bilanciamento su QuickFind

Per ridurre il costo dell'operazione di union, si considera come insieme primario quello con cardinalità maggiore, e modificare il padre delle foglie dell'altro insieme. Il valore **size(x)** memorizza la cardinalità dell'insieme x: quando viene svolta la union, si controlla size(x) per identificare l'insieme primario.

9.5 Bilanciamento su QuickUnion

Sono possibili 2 ottimizzazioni sulla QuickUnion:

- **Union by rank**: nell'unione degli insiemi A e B, rendiamo la radice dell'albero più basso figlia della radice dell'albero più alto
- **Union by size**: nell'unione degli insiemi A e B, rendiamo la radice dell'albero con meno nodi figlia della radice dell'albero con più nodi

9.5.1 QuickUnion by rank

Introduciamo un parametro **rank** che tenga conto dei livelli dell'albero. Quando facciamo una union, se $rank(B) < rank(A)$, rendiamo la radice di B figlio della radice di A; per $rank(A) < rank(B)$, viceversa.

Dimostriamo che l'albero con radice x ha almeno $2^{rank(x)}$ nodi.

Dimostrazione. **Caso base**. $rank(A) = 0$ ed un solo nodo. Allora $|A| = 1 \geq 2^0 = 2^{rank(A)}$

Passo. Dopo ogni union:

- se $rank(A) > rank(B)$: $|A \cup B| = |A| + |B| = 2^{rank(A \cup B)}$
- se $rank(B) > rank(A)$: simmetrico al precedente
- se $rank(A) = rank(B)$: $|A \cup B| = |A| + |B| = 2^{rank(A \cup B)}$

Il numero massimo di nodi è n, quindi $2^{rank(x)} \leq n$ e $rank(x) \leq \log_2 n$. La find richiede tempo $\mathcal{O}(\log n)$. □

9.5.2 QuickUnion by size

Utilizziamo il parametro $size(x)$ =numero di nodi dell'albero di cui x è radice. Quando eseguiamo una union, se $size(B) \leq size(A)$ rendiamo la radice di B figlio della radice di A, invece se $size(B) < size(A)$ rendiamo la radice di A figlio della radice di B, e scambiamo le due radici. Infine, $size(A) = size(A) + size(B)$.

9.6 Compressione nell'operazione find

Le euristiche di compressione vengono applicate durante l'esecuzione di una find, ed hanno lo scopo di diminuire l'altezza dell'albero. Ne vediamo tre tipi:

- **path compression**: ad ogni find prendiamo il reiferimento di ogni nodo radice e poi li facciamo diventare figli dell'ultimo nodo radice
- **path splitting**: rendo un nodo figlio di suo nonno
- **path halving**: come path halving ma con un solo puntatore

Combinando le euristiche di bilanciamento e compressione, una qualunque sequenza di n makeSet, m find e $n-1$ union può essere eseguita in $\mathcal{O}((n+m)log^*)$.

10