

Indice

| | | |
|----------|--|-----------|
| 1 | Grafi come strutture dati | 1 |
| 1.1 | Introduzione e terminologia | 1 |
| 1.2 | Rappresentazione | 2 |
| 2 | Visite | 5 |
| 2.1 | Visita generica | 5 |
| 2.2 | Visita in ampiezza | 8 |
| 2.3 | Visita in profondità | 10 |
| 3 | Aciclicità | 13 |
| 3.1 | Grafi orientati | 13 |
| 3.2 | Grafi non orientati | 13 |
| 3.3 | Test di aciclicità | 14 |
| 4 | Ordinamento topologico | 15 |
| 4.1 | Raggiungibilità | 15 |
| 4.2 | Ordine topologico | 15 |
| 5 | Componenti connesse e fortemente connesse | 17 |
| 5.1 | Componenti connesse | 17 |
| 5.2 | Componenti fortemente connesse | 18 |
| 6 | Tecnica greedy | 21 |
| 6.1 | Problemi | 21 |
| 6.2 | Problema dello zaino frazionario | 21 |
| 6.3 | Sottostruttura ottima e scelta greedy | 22 |
| 6.4 | Schema di algoritmo greedy | 22 |
| 6.5 | Massimo numero di intervalli disgiunti | 22 |
| 6.6 | Algoritmo di Moore | 23 |
| 6.7 | Codici di Huffman | 25 |
| 7 | Tecnica Greedy applicata ai grafi | 29 |
| 7.1 | Algoritmo di Dijkstra | 29 |
| 7.2 | Minimo albero ricoprente | 32 |
| 7.3 | Algoritmo di Prim | 33 |
| 7.4 | Algoritmo di Kruskal | 34 |

| | | |
|-----------|---|-----------|
| 8 | Programmazione dinamica | 37 |
| 8.1 | Tecnica della programmazione dinamica | 37 |
| 8.2 | Longest Common Subsequence | 38 |
| 8.3 | Zaino 0-1 | 39 |
| 9 | UnionFind | 41 |
| 9.1 | Definizione del problema | 41 |
| 9.2 | Implementazione | 41 |
| 9.3 | Complessità e costo ammortizzato | 42 |
| 9.4 | Bilanciamento su QuickFind | 42 |
| 9.5 | Bilanciamento su QuickUnion | 43 |
| 9.6 | Compressione nell'operazione find | 43 |
| 10 | Programmazione dinamica applicata ai grafi | 45 |
| 10.1 | Algoritmo di Bellman-Ford | 45 |
| 10.2 | Algoritmo di Floyd-Warshall | 48 |
| 11 | Teoria della complessità | 51 |
| 11.1 | Definizione | 51 |
| 11.2 | Problemi | 51 |
| 11.3 | Classi di complessità | 51 |
| 11.4 | Inclusione propria | 52 |
| 11.5 | Non determinismo | 53 |
| 11.6 | P vs NP | 54 |
| 11.7 | Problemi NP-completi | 54 |
| 11.8 | Problemi indecidibili | 54 |
| 12 | Algoritmi di approssimazione | 57 |
| 12.1 | Copertura dei vertici | 57 |
| 12.2 | Problema del commesso viaggiatore | 58 |
| 12.3 | Ricerca locale | 58 |

Algoritmi

| | | |
|----|--|----|
| 1 | INIZIALIZZA(G) | 5 |
| 2 | VISITA(G,s) | 5 |
| 3 | VISITA(G,s) | 6 |
| 4 | VISITA TUTTI I VERTICI(G) | 7 |
| 5 | VISITA(G,s) | 7 |
| 6 | VISITA BFS(G,s) | 8 |
| 7 | VISITA DFS (ottimizzata) | 10 |
| 8 | VISITA DFS RICORSIVA(G,u) | 10 |
| 9 | CICLICO(G) | 14 |
| 10 | VISITA RICORSIVA CICLO(G,u) | 14 |
| 11 | TOPOLOGICAL SORT(G) | 16 |
| 12 | TOPOLOGICAL DFS(G,u,ord,t) | 16 |
| 13 | COMPONENTI CONNESSE(G) | 17 |
| 14 | CONNESSIONE(G) | 18 |
| 15 | GREEDY(A) | 22 |
| 16 | MOORE() | 24 |
| 17 | HUFFMAN(C,f) | 26 |
| 18 | DIJKSTRA(G,W,s) | 29 |
| 19 | DIJKSTRA CON PRIORITY QUEUE(G,W,s) | 30 |
| 20 | TROVA MAR(G) | 33 |
| 21 | PRIM(G,W,s) | 34 |
| 22 | KRUSKAL CON UNIONFIND(G) | 35 |
| 23 | MSI(n,A) | 38 |
| 24 | LCS(a,b,m,n) | 39 |
| 25 | ZAINO(n,P,v[],p[]) | 40 |
| 26 | BELLMAN-FORD(G,W,s) | 46 |
| 27 | BF-NEG(G,W,s) | 47 |
| 28 | BF-DAG(G,W,s) | 47 |
| 29 | FLOYD-WARSHALL(G,W) | 49 |
| 30 | PATH RECONSTRUCTION(P,x,y) | 49 |
| 31 | SODDISFACIBILITÀ(formula,assegnazioni(z),i) | 53 |
| 32 | QUANTIFICATA($\xi_i x_i, \dots, \xi_n x_n : e(x_1, x_2, \dots, x_n), z_1, z_2, \dots, z_{i-1}, i$) | 53 |
| 33 | G(prog) | 55 |
| 34 | APPROX-COVER(G) | 57 |
| 35 | APPROX-TSP(G,W,r) | 58 |

Capitolo 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

Diamo un po' di definizioni:

- 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 $W : E \rightarrow \mathbb{R}$; 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 $\langle w_1, w_2, \dots, w_n \rangle$ 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 rappresentazione, bisogna considerare lo *spazio* occupato dalla struttura dati e il *costo computazionale* delle operazioni da effettuare su di essa.

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 $\mathcal{O}(m)$.

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).

Liste di incidenza

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

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, $v \rightarrow_1 w$ se e solo se $M[v, w] \neq 0$: moltiplicando la matrice per se stessa, il risultato è diverso da 0 solo se esiste un cammino di lunghezza 2 (e via dicendo).

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.

Capitolo 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.

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  $\leftarrow$  vettore di lunghezza  $n$ 
for ogni  $u \in V$  do
    color[ $u$ ]  $\leftarrow$  white
end for
```

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

Algoritmo 2 VISITA(G, s)

```
INIZIALIZZA( $G$ )
color  $\leftarrow$  gray
{visita  $s$ }
while ci sono vertici grigi do
     $u \leftarrow$  scegli un vertice grigio
```

```

if esiste  $v$  bianco adiacente ad  $u$  then
    color[ $v$ ]  $\leftarrow$  gray
    {visita  $v$ }
else color[ $v$ ]  $\leftarrow$  black
end if
end while

```

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

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. \square

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  $\leftarrow$  gray
{visita  $s$ }
while ci sono vertici grigi do
     $u \leftarrow$  scegli un vertice grigio
    if esiste  $v$  bianco adiacente ad  $u$  then
        color[ $v$ ]  $\leftarrow$  gray
         $\pi[v] \leftarrow u$ 
        {visita  $v$ }
    else color[ $v$ ]  $\leftarrow$  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  $u \in V$  do
    if color[ $u$ ] = white then
        VISITA(G, $u$ )
    end if
end for

```

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[ $s$ ]  $\leftarrow$  gray
{visita  $s$ }
Add(D, $s$ )
while NotEmpty(D) do
     $u \leftarrow First(D)$ 
    if esiste  $v$  bianco adiacente ad  $u$  then
        color[ $v$ ]  $\leftarrow$  gray

```

```

         $\pi[v] \leftarrow u$ 
        {visita  $v$ }
        Add(D, $v$ )
    else
        color[ $v$ ]  $\leftarrow$  black
        RemoveFirst(D)
    end if
end while

```

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

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 coda (il vertice in testa rimane nella coda finché non diventa nero).

Algoritmo 6 VISITA BFS(G, s)

```

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

```

```

    else
        color[ $v$ ]  $\leftarrow$  black
        dequeue( $D$ )
    end if
end while

```

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 .

Proprietà

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

Proprietà (2). *Se $\langle v_1, v_2, \dots, v_n \rangle$ 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 $\langle v_2, \dots, v_n \rangle$, 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. $d[v] = d[v_1] + 1 \leq d[v_1] + 1$

□

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 $V_k = \{v \in V \mid \delta(s, v) = k\}$ (v_0 contiene solo la sorgente).

Nel caso base, $d[v_0] \leq \delta(s, v)$ (distanza di s da sé 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 w a v . Definiamo l'insieme dei vertici appartenenti a V_{k-1} con arco entrante in v come $U_{k-1} = \{w \in V_{k-1} \mid (w, v) \in E\}$. 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 (perché 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 $d[v] \leq (k - 1) + 1$
- $d[v] \leq k$

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*. \square

2.3 Visita in profondità

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 NotEmpty(D) do
    while esiste  $v$  non considerato adiacente a top(D) do
        if color[v] = white then
            color[v] = gray
             $\pi[v] \leftarrow \text{top}(D)$ 
            {visita s}
            push(D, s)
        end if
    end while
    color[top(D)] ← black
    pop(D)
end while

```

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
         $\pi[v] \leftarrow u$ 
        VISITA DFS RICORSIVA( $G, v$ )
    end if
    color[u] ← black
    f[u] ← time
    time++
end for

```

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).

Capitolo 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, $\text{color}[v]$ 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, $\text{color}[v]$ 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 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  $\text{color}[u] = \text{white}$  and VISITA RICORSIVA CICLO(G,u) then
        return true
    end if
end for
return false

```

Algoritmo 10 VISITA RICORSIVA CICLO(G,u)

```

 $\text{color}[u] \leftarrow \text{gray}$ 
for ogni  $v$  adiacente ad  $u$  do
    if  $\text{color}[v] = \text{white}$  then
         $\pi[v] \leftarrow u$ 
        if VISITA RICORSIVA CICLO(G,v) then
            return true
        end if
    else if  $v \neq \pi[u]$  ( $\text{color}[v] = \text{gray}$  per grafi orientati) then
        return true
    end if
end for
 $\text{color}[u] \leftarrow \text{black}$ 
return false

```

Capitolo 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 raggiungibile 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.

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} . \square

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

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. \square

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  $\leftarrow$  vettore di lunghezza n
t  $\leftarrow$  n-1
for ogni  $u \in V$  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]  $\leftarrow$  gray
d[u]  $\leftarrow$  time  $\leftarrow$  time+1
for ogni  $v$  adiacente ad  $u$  do
    if color[v] = white then
         $\pi_{getsu}$ 
        TOPOLOGICAL DFS(G,v,ord,t)
    end if
end for
color[u]  $\leftarrow$  black
f[u]  $\leftarrow$  time  $\leftarrow$  time+1
ord[t]  $\leftarrow$  u
t-

```

La complessità è $\mathcal{O}(m + n)$.

Capitolo 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 degenero 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( $G$ )
 $\Pi \leftarrow$  foresta vuota
for ogni  $u \in V$  do
    if  $\text{color}[u] = \text{white}$  then
         $\pi_u \leftarrow \text{VISITA}(G, u)$  ( $\pi_u$  è l'albero di visita)
         $\Pi \leftarrow \Pi + \pi_u$ 
    end if
end for
return  $\Pi$ 
```

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  $G$ 
 $\pi \leftarrow \text{VISITA}(G, s)$ 
for ogni  $u$  appartenente a  $G$  do
    if  $u \neq s$  e  $\pi[u] = \text{NULL}$  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 ($D(x)$), ossia i vertici di G raggiungibili da x
- calcoliamo gli *antenati* di x ($A(x)$), ossia i vertici che raggiungono x
- $\text{cfc}[x]$ 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 $x \rightarrow z$ e $z \rightarrow y$. z è banalmente raggiungibile da x per ipotesi ($x \rightarrow z$); siccome x, y appartengono alla stessa cfc, esisterà un cammino $y \rightarrow x$. Esiste anche $z \rightarrow y$ per ipotesi. Quindi, per concatenazione, esisterà anche un cammino $z \rightarrow y \rightarrow x$. \square

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. \square

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 esisterà 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 partire 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 . \square

L'algoritmo è quindi corretto.

Capitolo 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 valutata(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

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_i , $0 \leq x_i \leq 1$. 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?

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. attribuisce 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

Algoritmo 15 GREEDY(A)

```

 $S \leftarrow$  insieme vuoto
while  $S$  non è una soluzione do
    estrai da  $A$  l'elemento più appetibile  $a$ 
    if  $S \cup a$  è ammissibile then
        aggiungi  $a$  all'insieme  $S$ 
    end if
end while

```

possono essere fisse o modificabili.

6.5 Massimo numero di intervalli disgiunti

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. ordina 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, $\mathcal{O}(n \log n)$.

Correttezza

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

- *Max*: la sequenza A_1, A_2, \dots, A_k di intervalli disgiunti scelti è, per l'insieme S , una *sequenza massimale*
- *PrimaMax*: la sequenza A_1, A_2, \dots, 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, \dots, 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, \dots, A_k ; allora, la sequenza A_1, A_2, \dots, 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

Problema dello scheduling e algoritmo risolvibile

Una *sequenza di scheduling*, o semplicemente *scheduling*, è una sequenza di lavori L_1, L_2, \dots, 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
     $t = t + \text{durata di } L_i$ 
    if  $t > \text{scadenza di } L_i$  then
        toglì da Sol il lavoro  $L_{max}$  di durata massima
         $t = t - \text{durata di } L_{max}$ 
    end if
end for

```

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, \dots, 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 $Sol = L_1, L_2, \dots, L_k$. 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, \dots, 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, \dots, L_k è uno scheduling massimale e di durata minima. \square

6.7 Codici di Huffman

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 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*.

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.

Algoritmo di Huffman

L'algoritmo di Huffman ha come input: un alfabeto, cioè un insieme C di caratteri; una funzione f che dà 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 :

$$L(T) = \sum_{c \in C} d_c \cdot f(c)$$

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:

$$B(T) = \sum_{c \in C} d_c \cdot n \cdot f(c) = n \cdot L(T)$$

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 finché si ottiene un unico albero

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

Algoritmo 17 HUFFMAN(C, f)

```

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

```

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, \dots, T_n sono sottoalberi di un albero di Huffman T per quell'alfabeto.

Invariante. La foresta $\{T_1, T_2, \dots, T_n\}$ 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, \dots, 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 $F = \{T_1, T_2, \dots, T_n\}$ 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, \dots, 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 $T_1, \dots, T_a, \dots, T_b, \dots, T_n$ 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 $f(T_a) \leq f(T_b)$ e $f(T_x) \leq f(T_y)$, abbiamo $f(T_a) \leq f(T_x)$ e $f(T_b) \leq f(T_y)$. 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, $d_1 \leq d$ e $d_2 \leq d$.

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 an-

che 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. \square

Capitolo 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( $G$ )
 $color[s] \leftarrow gray$ 
 $d[s] \leftarrow 0$ 
while esistono vertici grigi do
     $u \leftarrow$  vertice grigio con  $d[u]$  minore
     $S \leftarrow S \cup \{u\}$ 
    for ogni  $v$  adiacente ad  $u$  do
        if  $color[v] \neq black$  then
             $color[v] \leftarrow gray$ 
            if  $d[v] > d[u] + W(u, v)$  then
                 $\pi[v] \leftarrow u$ 
                 $d[v] \leftarrow d[u] + W(u, v)$ 
            end if
        end if
    end for
```

Miglioramenti

I nodi grigi sono gestiti con una *coda di priorità*. È possibile non distinguere nodi bianchi e grigi ed inserire tutti nella coda fin dall'inizio, assegnando loro distanza infinita da s : i nodi neri sono

```

    color[u] ← black
end while

```

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( $G$ )
 $D \leftarrow$  coda di priorità vuota
 $d[s] \leftarrow 0$ 
for ogni  $v$  in  $v[G]$  do
    enqueue( $D, v, d[v]$ )
end for
while NotEmpty( $D$ ) do
     $u \leftarrow$  dequeueMin( $D$ )
     $S \leftarrow S \cup \{u\}$ 
    for ogni  $v$  adiacente ad  $u$  do
        if  $d[v] > d[u] + W(u, v)$  then
             $\pi[v] \leftarrow u$ 
             $d[v] \leftarrow d[u] + W(u, v)$ 
            decreaseKey( $D, v, d[v]$ )
        end if
    end for
end while

```

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*: $\mathcal{O}(n + n * m)$

Versione di Johnson

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

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 $s \equiv v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{k-1} \rightarrow v_k \equiv u$ 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 $d[s] = 0$. \square

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'iniziazione 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 $\pi[u] = r \neq \text{NULL}$ 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 $W(s, \dots, x, \dots, y, \dots, u) = d[y] + W(y \rightsquigarrow u)$ 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 . \square

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 miglior 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

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.

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 . \square

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 .

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

Algoritmo 20 TROVA MAR(G)

```

 $A \leftarrow$  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

```

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 ad 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. \square

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 .

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 $d[v] \leftarrow W(u, v)$.

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 $\mathcal{O}(m \log n)$.

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

Algoritmo 21 PRIM(G, W, s)

```

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

```

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 . \square

7.4 Algoritmo di Kruskal

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

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) , si considera $\text{find}(u)$ e $\text{find}(v)$: se sono uguali, u e v appartengono allo stesso albero.

Algoritmo 22 KRUSKAL CON UNIONFIND(G)

```

 $A \leftarrow$  insieme di archi vuoto
ordina gli archi in una sequenza  $S$  in ordine non decrescente di peso
crea una UnionFind contenente i vertici di  $G$  come insiemi iniziali
for ogni arco  $(u, v) \in S$  do
    if  $\text{find}(u) \neq \text{find}(v)$  then
        aggiungi  $(u, v)$  ad  $A$ 
        union( $u, v$ )
    end if
end for
  
```

Complessità

La complessità è pari a $\mathcal{O}(m \log n)$, come l'algoritmo di Prim.

Correttezza

Invariante Gli archi in A definiscono una foresta che è sottoinsieme di un certo MAR.

Caso base L'invariante è banalmente vero all'istante iniziale, quando A non contiene nessun arco.

Passo

Caso 1 Si sceglie (u, v) come l'arco più leggero non ancora considerato. Nel primo caso, u e v appartengono allo stesso albero; l'aggiunta di (u, v) creerebbe un ciclo, quindi viene scartato.

Caso 2 Nel secondo caso, u e v appartengono a due alberi distinti T_1 e T_2 . Consideriamo un taglio che non tagli nessun arco di A , e che abbia T_1 e T_2 da parti opposte, tagliando l'arco (u, v) . L'arco (u, v) è l'arco di peso minimo fra tutti gli archi che attraversano il taglio. Quindi, per il lemma del taglio, sappiamo che $(u, v) \cup A$ è un sottoinsieme di un MAR.

Capitolo 8

Programmazione dinamica

8.1 Tecnica della programmazione dinamica

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.

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). \square

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 $S_{i-2} \cup \{V_i\}$.*

$$\begin{aligned}
S_0 &= \emptyset \\
S_1 &= w_1 \\
S_i &= \begin{cases} S_{i-1} & \text{se } W(S_{i-1}) > W(S_{i-1}) + w_i \\ S_{i-2} \cup V_i & \text{altrimenti} \end{cases}
\end{aligned}$$

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] ← {}
A[1] ← {V1}
for i=2...n do
    A[i] ← maxPeso(A[i-1], A[i-2] + Vi)
end for
return A[n]

```

8.2 Longest Common Subsequence

Data una sequenza $S : a_1, \dots, a_m$, 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.

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: $S3 = lcs(S1, S2)$.

Sottostruttura ottima

Siano $S1 : a_1, \dots, a_m$ e $S2 : b_1, \dots, b_n$ e $S3 : c_1, \dots, c_k$; 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 dire 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 $lcs(S1, S2 - \{b_n\})$.

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.

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 precedente.

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

$LCS \leftarrow$ nuova matrice di dimensioni $m + 1 * n + 1$

$L \leftarrow$ nuova matrice di dimensioni $m + 1 * n + 1$

for $i = 0 \dots m$ **do**

$LCS[i, 0] \leftarrow []$

$L[i, 0] \leftarrow 0$

end for

for $i = 1 \dots m$ **do**

for $j = 1 \dots n$ **do**

if $a[i - 1] == b[j - 1]$ **then**

$LCS[i, j] \leftarrow \nwarrow$

$L[i, j] \leftarrow L[i - 1, j - 1] + 1$

else if $L[i - 1, j] > L[i, j - 1]$ **then**

$LCS[i, j] \leftarrow \uparrow$

$L[i, j] \leftarrow L[i - 1, j]$

else

$LCS[i, j] \leftarrow \leftarrow$

$L[i, j] \leftarrow L[i, j - 1]$

end if

end for

end for

return $LCS[m, n]$

Complessità

La 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.

Costruzione di algoritmi di programmazione dinamica

Passi da seguire per costruire 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

Costruzione zaino 0-1

Equazione ricorsiva per l'algoritmo:

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

$V(i, j)$ ha due parametri: i è l'ultimo oggetto considerato, j è la capienza. La struttura di memoizzazione è quindi una matrice; la soluzione sarà contenuta 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 ... P 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]
```

Capitolo 9

UnionFind

9.1 Definizione del problema

Il problema è mantenere una collezione di insiemi disgiunti sulla quale siano possibili 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*)

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 foglia di B , sostituisce il puntatore al padre con un puntatore alla radice di A e cancella la radice di B . $\mathcal{O}(n)$.

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*.

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

QuickUnion by rank

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

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

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

Passo. Dopo ogni union:

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

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

QuickUnion by size

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

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 $\mathcal{O}((n + m)\log^*)$.

Capitolo 10

Programmazione dinamica applicata ai grafi

10.1 Algoritmo di Bellman-Ford

Condizione di Bellman

Per ogni arco (u, v) e per ogni vertice s si ha $\delta(s, v) \leq \delta(s, u) + W(u, v)$, per la *disuguaglianza triangolare delle distanze*.

Lemma. *Un arco (u, v) appartiene ad un cammino minimo da s a v se e solo se u è raggiungibile da s e la condizione di Bellman è soddisfatta con l'uguaglianza per (u, v) , cioè se vale $\delta(s, v) = \delta(s, u) + W(u, v)$.*

Tecnica del rilassamento

Consideriamo una stima $D(s, v)$ per eccesso di $\delta(s, v)$, che sia il peso di un cammino sul grafo tra s e v (quindi, $D(s, v) \geq \delta(s, v)$). Supponiamo esista un arco (u, v) per cui valga $D(s, v) > D(s, u) + W(u, v)$. Per rendere vera la disuguaglianza triangolare, possiamo decrementare $D(s, v)$ con l'assegnazione $D(s, v) \leftarrow D(s, u) + W(u, v)$: questa operazione è detta *rilassamento dell'arco*.

Applicando esaustivamente l'operazione di rilassamento, per grafi senza cicli negativi, $D(s, v)$ converge a $\delta(s, v)$. Nel momento in cui $D(s, v) = \delta(s, v)$, l'ultimo arco per cui è stato applicato il rilassamento è quello per cui la condizione di Bellman vale con l'uguaglianza e (u, v) appartiene ad un cammino minimo tra s e v .

Programmazione dinamica per Bellman-Ford

Proprietà. *Un sottocammino di un cammino minimo è un cammino minimo.*

Sia $\langle s, v_1, v_2, \dots, v_k \rangle$ un cammino minimo da s a v_k . Grazie alla condizione di Bellman, possiamo scrivere: $\delta(s, v_k) = \delta(s, v_{k-1}) + W(v_{k-1}, v_k)$.

Sappiamo che il cammino minimo $\langle s, v_1, v_2, \dots, v_{k-1} \rangle$ avrà lunghezza $k - 1$, quindi il sottoproblema di trovare $\langle s, v_1, v_2, \dots, v_{k-1} \rangle$ dovrà considerare solo cammini di lunghezza $k - 1$ e così via fino a $k = 0$.

Usiamo le stime $D(s, v)$ come struttura di memorizzazione, ed inizializziamo $D(s, s) = 0$ e $D(s, v) = \infty$ per tutti gli altri vertici v del grafo, rispettando la condizione per cui $D(s, v) \geq \delta(s, v)$.

Applicando il rilassamento su tutti gli archi una sola volta, troveremo: $D(s, v_1) > D(s, s) + W(s, v_1)$ (poichè $W(s, v_1) < \infty$). Il rilassamento porrà $D(s, v_1) = D(s, s) + W(s, v_1)$. Ma $D(s, s) = \delta(s, s)$, e $\langle s, v_1 \rangle$ è un cammino minimo, quindi $D(s, v_1) = \delta(s, s) + W(s, v_1) = \delta(s, v_1)$.

Iterando il processo, al k -esimo passo avremo $D(s, v_k) = \delta(s, v_k)$. Un cammino minimo semplice può contenere al massimo tutti i vertici del grafo, quindi $n - 1$ archi. Di conseguenza, bastano $n - 1$ iterazioni per trovare i cammini minimi da s a tutti i vertici del grafo.

Algoritmo 26 BELLMAN-FORD(G, W, s)

```

INIZIALIZZA( $G$ )
 $d[s] \leftarrow 0$ 
for  $i = 1 \dots n - 1$  do
    for ogni  $(u, v)$  in  $G$  do
        if  $d[v] > d[u] + W(u, v)$  then
             $\pi[v] \leftarrow u$ 
             $d[v] \leftarrow d[u] + W(u, v)$ 
        end if
    end for
end for

```

Correttezza

Sia $\delta(s, v_k)$ a distanza da s a v_k .

Definiamo l'invariante *KPATH*: dopo la k -esima iterazione del ciclo esterno, si ha $d[v_k] = \delta(s, v_k)$ per ogni nodo v_k per cui il cammino minimo da s a v_k è composto al più da k archi.

Dimostrazione. Base. per $k = 0$, per il cammino da s a s , $d[s] = 0$ e $d[u] = \infty$ per tutti gli altri nodi u . *Passo.* Dimostriamo che *KPATH* sia vero anche dopo $k + 1$ iterazioni. Sia v_{k+1} un nodo il cui cammino minimo $s \rightsquigarrow v_k \rightsquigarrow v_{k+1}$ è composto da $k + 1$ archi. Sappiamo che $\delta(s, v_{k+1}) = \delta(s, v_k) + W(v_k, v_{k+1})$ e $d[v_k] = \delta(s, v_k)$. Quindi: $\delta(s, v_{k+1}) = d[v_k] + W(v_k, v_{k+1})$.

Al passo $k + 1$, ci sono due casi:

- $d[v_{k+1}]$ viene aggiornata, allora $d[v_{k+1}] = \delta(s, v_{k+1})$
- $d[v_{k+1}]$ non viene aggiornata, poichè $d[v_{k+1}]$ è il peso di un cammino da s a v_{k+1} in G , $d[v_{k+1}] \geq \delta(s, v_{k+1})$, e poichè non è stata aggiornata, $d[v_{k+1}] = \delta(s, v_{k+1})$.

I cammini restituiti dall'algoritmo sono effettivamente quelli minimi. □

Complessità

L'algoritmo fa $n - 1$ iterazioni del ciclo esterno. Per ogni iterazione, l'algoritmo considera tutti gli archi del grafo con una serie di operazioni di costo $\mathcal{O}(1)$. La complessità è quindi $\mathcal{O}(n * m)$

L'algoritmo di Dijkstra è più efficiente perchè sceglie (u, v) garantendo che (u, v) sia l'arco che minimizza $D(s, u) + W(u, v)$ tra tutti gli archi nei quali u è stato raggiunto da un cammino minimo e v no, compiendo $\mathcal{O}(m)$ operazioni di rilassamento; invece l'algoritmo di Bellman-Ford sceglie (u, v) tra tutti gli archi presenti in G .

Trovare cicli negativi

Se un cammino minimo contiene vertici ripetuti, vuol dire che nel grafo esiste un ciclo. Se il ciclo ha *peso positivo*, il cammino minimo è quello ottenuto percorrendolo una sola volta. Se il ciclo ha *peso 0*, viene rilevato il cammino minimo ottenuto percorrendo il ciclo una volta sola. Se il ciclo ha *peso negativo*, ogni volta che viene percorso viene trovato un cammino minimo più lungo e di peso minore: per i cicli negativi non sarà mai possibile trovare una soluzione ottima. Per rilevare cicli

Algoritmo 27 BF-NEG(G, W, s)

```

INIZIALIZZA( $G$ )
 $d[s] \leftarrow 0$ 
for  $i = 1 \dots n - 1$  do
    for ogni  $(u, v)$  in  $G$  do
        if  $d[v] > d[u] + W(u, v)$  then
             $\pi[v] \leftarrow u$ 
             $d[v] \leftarrow d[u] + W(u, v)$ 
        end if
    end for
end for
for ogni  $(u, v)$  in  $G$  do
    if  $d[v] > d[u] + W(u, v)$  then
return errore
    end if
end for

```

negativi basta controllare se l' n -esima iterazione aggiorna qualche distanza: se così succede, esiste un ciclo negativo.

Ottimizzazione di BF per DAG

L'algoritmo di Bellman-Ford compie molti passi di rilassamento inutili. In particolare, l'assegnazione $D(s, v_k) \leftarrow D(s, v_{k-1}) + W(v_{k-1}, v_k)$ è inutile se $D(s, v_{k-1}) > \delta(s, v_{k-1})$: infatti, sarà poi necessario almeno un ulteriore rilassamento dello stesso arco per calcolare $\delta(s, v_k)$. Se un grafo è un DAG, si può calcolare prima un ordinamento topologico, e poi passare tutti gli archi una sola volta seguendo l'ordine topologico. Ha complessità $\mathcal{O}(m + n)$.

Algoritmo 28 BF-DAG(G, W, s)

```

INIZIALIZZA( $G$ )
 $d[s] \leftarrow 0$ 
 $ord \leftarrow \text{TOPOLOGICAL-SORT}(G)$ 
for  $i = 1 \dots n - 1$  do
    for ogni  $(ord[i], v)$  do
        if  $d[v] > d[ord[i]] + W(ord[i], v)$  then
             $\pi[v] \leftarrow ord[i]$ 
             $d[v] \leftarrow d[ord[i]] + W(ord[i], v)$ 
        end if
    end for
end for

```

10.2 Algoritmo di Floyd-Warshall

Distanze e cammini minimi k-vincolati

Denotiamo i vertici del grafo come v_1, v_2, \dots, v_n . Per un k fissato con $1 \leq k \leq n$ definiamo: *cammino minimo k-vincolato* tra x e y (denotato con π_{xy}^k) il cammino che va da x a y di costo minimo tra tutti quelli che non contengono i vertici $\{v_{k+1}, \dots, v_n\}$ e *distanza k-vincolata* (denotata con d_{xy}^k), che è il peso W se π_{xy}^k esiste, ∞ altrimenti.

Un cammino minimo k-vincolato è un cammino minimo in un *grafo k-vincolato* G_k : varrà quindi la proprietà della sottostruttura ottima e sarà possibile applicare tecniche di programmazione dinamica.

Per ogni $1 \leq k \leq n$ e per ogni coppia x, y definiamo l'equazione ricorsiva:

$$d_{xy}^k = \min(d_{xy}^{k-1}, d_{xv_k}^{k-1} + d_{v_k y}^{k-1})$$

In poche parole, preso un k , si va a vedere se la concatenazione dei cammini minimi da x a v_k e da v_k a y ha peso minore del cammino da x a y e si verifica la disuguaglianza triangolare: se *non* si verifica, si applica un rilassamento facendo passare il cammino minimo attraverso v_k . Abbiamo due casi:

- $v_k \notin \pi_{xy}^k$: allora π_{xy}^k è anche un cammino minimo $(k-1)$ vincolato; se così non fosse, esisterebbe un cammino minimo $(k-1)$ vincolato di peso minore, ma questo sarebbe anche un cammino k-vincolato con peso minore di π_{xy}^k (assurdo)
- $v_k \in \pi_{xy}^k$: i sottocammini da x a v_k e da v_k a y sono cammini minimi k-vincolati (per sottostruttura ottima); poichè non contengono internamente v_k , essi sono anche cammini $(k-1)$ vincolati.

Struttura di memoizzazione

Per modellare, per ogni coppia di vertici x e y , la lunghezza del cammino provvisorio tra i due vertici, per ogni k-esima iterazione dell'algoritmo, servono n matrici D^k di dimensione $n \times n$ (lo spazio richiesto è $\mathcal{O}(n^3)$).

$D^{k-1}[i, j]$ contiene la distanza tra il vertice v_i e v_j in un grafo G^{k-1} in cui $V^{k-1} = G - \{v_k, v_{k+1}, \dots, v_n\} \cup \{v_i, v_j\}$.

$D^k[i, j]$ contiene la distanza tra il vertice v_i e v_j in un grafo G^k in cui $V^k = G - \{v_{k+1}, \dots, v_n\} \cup \{v_i, v_j\}$.

Avendo aggiunto solo v_k rispetto al grafo precedente, le uniche strade diverse passano da v_k . Il miglior cammino da v_i a v_j che passa da v_k in G^k è la concatenazione del miglior cammino da v_i a v_k ed il migliore da v_k a v_j : questi sono già presenti nella matrice D^{k-1} .

In G^k , il miglior cammino da v_i a v_j è quello di distanza minore tra $D^{k-1}[i, j]$ e $D^{k-1}[i, k] + D^{k-1}[k, j]$.

Algoritmo di Floyd-Warshall

È possibile usare un'unica matrice, aggiornata ad ogni iterazione, in quanto $D^{k-1}[i, k]$ e $D^{k-1}[k, j]$ non cambieranno mai valore tra iterazioni.

Algoritmo 29 FLOYD-WARSHALL(G, W)

```

for  $i = 1 \dots n$  do
  for  $j = 1 \dots n$  do
    if  $i = j$  then
       $D[i, j] \leftarrow 0$ 
    else if  $(i, j) \in E$  then
       $D[i, j] \leftarrow W(i, j)$ 
    else
       $D[i, j] \leftarrow \infty$ 
    end if
  end for
end for
for ( $k = 1 \dots n$ )
  for  $i = 1 \dots n$  do
    for  $j = 1 \dots n$  do
      if  $D[i, j] > D[i, k] + D[k, j]$  then
         $D[i, j] \leftarrow D[i, k] + D[k, j]$ 
      end if
    end for
  end for
end for

```

Cicli negativi e path reconstruction

In presenza di un ciclo negativo è possibile raggiungere un vertice v da se stesso con un cammino di distanza minore di 0. La distanza da un vertice da se stesso è riportata nella diagonale della matrice, quindi basta aggiungere un controllo.

È possibile ricostruire il cammino minimo partendo da un vettore di predecessori.

Algoritmo 30 PATH RECONSTRUCTION(P, x, y)

```

if  $x = y$  then
  return  $x$ 
end if return PATH RECONSTRUCTION( $P, x, P[x, y]$ ) +  $y$ 

```

Capitolo 11

Teoria della complessità

11.1 Definizione

La complessità di un problema (non di un algoritmo) è la complessità del migliore algoritmo che lo risolve, sia esso noto oppure ipotizzato.

La complessità di un problema non dipende nè dal linguaggio di programmazione utilizzato, nè dal tipo di calcolatore (a patto che sia un modello di calcolo *Turing completo*).

11.2 Problemi

Un *problema* P è una relazione del tipo

$$P \subseteq I \times S$$

Dove I è l'insieme delle possibili istanze in ingresso ed S è l'insieme delle possibili soluzioni del problema. Esistono diversi tipi di problemi:

- *Problemi di decisione*: problemi che richiedono di verificare una certa proprietà sull'input.
- *Problemi di ricerca*: data l'istanza di un problema, restituire una soluzione ammissibile.
- *Problemi di ottimizzazione*: data l'istanza di un problema, restituire una soluzione ottima secondo un determinato criterio.

11.3 Classi di complessità

Una *classe di complessità* è un insieme di problemi che possono essere risolti usando le stesse risorse di calcolo. Dato n , che indica la dimensione dell'input, possiamo classificare i problemi in base al tempo o allo spazio richiesti:

- $TIME(f(n))$: è l'insieme dei problemi che possono essere risolti in tempo $\mathcal{O}(f(n))$
- $SPACE(f(n))$: è l'insieme dei problemi che possono essere risolti in spazio $\mathcal{O}(f(n))$

Classe P

Per semplicità, le classi di complessità sono raggruppate nelle cosiddette *classi unione*.

La classe P è la classe dei problemi risolvibili in tempo polinomiale rispetto ad n .

$$P = \bigcup_{c=0}^{\infty} TIME(n^c)$$

Ad esempio, i problemi risolvibili in $\mathcal{O}(n \log n)$ appartengono a P .

Classe PSPACE

La classe $PSPACE$ è la classe dei problemi risolvibili in spazio polinomiale rispetto ad n .

$$PSPACE = \bigcup_{c=0}^{\infty} SPACE(n^c)$$

Potendo fare al più un numero polinomiale di operazioni, e quindi anche di accessi in memoria, $P \subseteq PSPACE$.

Classe EXPTIME

La classe $EXPTIME$ è la classe dei problemi risolvibili in tempo esponenziale rispetto ad n .

$$EXPTIME = \bigcup_{c=0}^{\infty} TIME(2^{n^c})$$

Potendo fare al più un numero polinomiale di operazioni, e quindi anche di accessi in memoria, $P \subseteq PSPACE$.

11.4 Inclusione propria

L'unica inclusione propria dimostrata è $P \subset EXPTIME$, ovvero esiste almeno un problema che non può essere risolto in tempo polinomiale. Capire se $P \subset PSPACE$ e $PSPACE \subset EXPTIME$ sono problemi aperti.

Introduciamo un problema appartenente a $PSPACE$ per cui non esistono algoritmi risolvibili polinomiali, quello dell' *Espressione in Forma Normale Congiuntiva*: congiunzione di disgiunzioni (*clausole*) di variabili booleane (*letterali*). Ad esempio

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

Il problema della soddisfacibilità di espressioni di questo tipo appartiene a $PSPACE$. Non si conoscono algoritmi che lo risolvono in tempo polinomiale.

Identifichiamo con $e(x_1, x_2, \dots, x_n)$ la formula. L'algoritmo ha complessità $\mathcal{O}(2^n)$. Al termine dell'esecuzione, z_1, z_2, \dots, z_n è un'assegnazione che rende vera la formula.

Un *certificato* è un oggetto y , dipendente dall'istanza x e dal problema P , che possa giustificare $(x, 1) \in P$ (per i problemi decisionali). Gli algoritmi possono essere riscritti in due fasi: fase di *costruzione* e fase di *verifica*.

Algoritmo 31 SODDISFACIBILITÀ(formula,assegnazioni(z),i)

```

if  $i = n + 1$  then return  $e(z_1, z_2, \dots, z_n)$ 
else
   $v_0 \leftarrow$  SODDISFACIBILITÀ ( $e(x_1, x_2, \dots, x_n), z_1, z_2, \dots, z_{i-1}, 0, i + 1$ )
   $v_1 \leftarrow$  SODDISFACIBILITÀ ( $e(x_1, x_2, \dots, x_n), z_1, z_2, \dots, z_{i-1}, 1, i + 1$ )
end if
return  $v_0 \vee v_1$ 

```

11.5 Non determinismo

Definiamo una funzione $indovinaz \in \{0, 1\}$, che può indovinare, in tempo costante, un giusto valore booleano: con questa funzione, l'algoritmo precedente avrebbe complessità $\mathcal{O}(n + m)$, dove m è il numero di clausole.

Gli algoritmi sviluppati con funzioni di questo tipo sono detti *non deterministici*, perchè la loro computazione non è deterministica. La fase di verifica è deterministica.

Chiamiamo $NTIME(f(n))$ la classe di problemi risolvibili da un algoritmo non deterministico in tempo $\mathcal{O}(f(n))$. La classe NP è la classe di problemi risolvibili in tempo polinomiale da un algoritmo non deterministico.

$$NP = \bigcup_{c=0}^{\infty} NTIME(n^c)$$

Un algoritmo deterministico è un caso particolare di un algoritmo non deterministico, quindi $P \subseteq NP$. Inoltre, il certificato deve occupare spazio polinomiale, quindi $NP \subseteq PSPACE$.

Si sospetta, ma non è noto, che $P \subset NP$ e $NP \subset PSPACE$. Introduciamo un problema appartenente a $PSPACE$ ma che si sospetta non appartenere a NP , quello della *Formula Booleana Quantificata*: è un'espressione in forma normale congiuntiva preceduta da una serie di quantificatori universali ed esistenziali che legano tutte le variabili. Ad esempio,

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_4})$$

Data tale formula, il problema richiede di verificare se essa è vera.

Identifichiamo con $e(x_1, x_2, \dots, x_n)$ la formula e $\xi_i x_i$ i quantificatori.

Algoritmo 32 QUANTIFICATA($\xi_i x_i, \dots, \xi_n x_n : e(x_1, x_2, \dots, x_n), z_1, z_2, \dots, z_{i-1}, i$)

```

if  $i = n + 1$  then return  $e(z_1, z_2, \dots, z_n)$ 
else
   $v_0 \leftarrow$  QUANTIFICATA ( $\xi_i x_i, \dots, \xi_n x_n : e(x_1, x_2, \dots, x_n), z_1, z_2, \dots, z_{i-1}, 0, i + 1$ )
   $v_1 \leftarrow$  QUANTIFICATA ( $\xi_i x_i, \dots, \xi_n x_n : e(x_1, x_2, \dots, x_n), \dots, z_{i-1}, 1, i + 1$ )
  if  $\xi_i = \exists$  then return  $v_0 \vee v_1$ 
  else return  $v_0 \wedge v_1$ 
end if
end if

```

L'algoritmo ha complessità $\mathcal{O}(2^n)$.

Il certificato da verificare ha esso stesso dimensione esponenziale, così come la sua verifica.

11.6 P vs NP

Il confine tra P e NP è quello più importante nella teoria della complessità: infatti, è quello che separa problemi "trattabili" da quelli "non trattabili". Troviamo un modo per confrontare i problemi.

Diciamo che un problema $P1 \subseteq I1 \times \{0, 1\}$ è riconducibile polinomialmente ad un problema $P2 \subseteq I2 \times \{0, 1\}$ se e solo se:

- esiste una funzione che, data un'istanza $x1 \in P1$, la trasforma in un'istanza $x2 \in P2$ in tempo polinomiale
- per ogni coppia di istanze $x1$ e $x2$ così costruite, la soluzione di $x1$ è la stessa soluzione di $x2$

Proprietà. Se $P2$ appartiene alla classe P , anche $P1$ appartiene a P .

Ad esempio, il problema del Ciclo Hamiltoniano ($P1$) è riconducibile polinomialmente al problema del Cammino Semplice ($P2$).

Si possono definire le seguenti classi:

- *Problemi NP-hard*: un problema decisionale D si definisce NP-hard se ogni problema $Q \in NP$ è riconducibile a D in tempo polinomiale
- *Problemi NP-completi*: un problema decisionale D si definisce NP-completo se è NP-hard ed appartiene alla classe NP .

11.7 Problemi NP-completi

Un problema NP-completo è quello dell'*Halt limitato*: dato un programma X ed un intero k , verificare se esiste un input per cui X termina in al massimo k passi.

Appartiene ad NP perchè la fase di verifica può essere effettuata in tempo polinomiale simulando $X(i)$ per k passi.

Ogni problema appartenente ad NP è riconducibile polinomialmente ad halt limitato.

Sia $Prob \in NP$, allora esiste un programma C che verifica un certificato in tempo polinomiale $p(n)$. Allora è possibile costruire un programma C' che va in loop ogniqualvolta la risposta di C sia negativa, e termina altrimenti. È possibile riscrivere $Prob$ come il problema dell'halt limitato, con C' come programma X e $p(n)$ come k . L'halt limitato restituirà 1 se esistono dei dati per cui C' termina in $p(n)$ passi, 0 altrimenti.

Il problema dell'halt limitato è NP e NP-hard, quindi è NP-completo.

11.8 Problemi indecidibili

Esistono una serie di problemi definiti *indecidibili*: non solo non si conoscono algoritmi per risolverli, ma è stato dimostrato che tali algoritmi non possono esistere, nemmeno utilizzando tempo e spazio infiniti.

Il problema dell'halt è un problema indecidibile.

Definire un programma $halt(P, i)$ che restituisca 1 se il programma $Prog$ (con un certo input i) termina in un numero finito di passi, 0 altrimenti.

Se potessimo scrivere halt, potremmo scrivere un programma G come:

Algoritmo 33 $G(\text{prog})$

```
if halt(Prog,Prog) then  
    loop  
else return 0  
end if
```

$G(G)$ termina solo se $G(G)$ non termina, che è una contraddizione.

Capitolo 12

Algoritmi di approssimazione

È sempre possibile associare ad un problema di ottimizzazione un problema decisionale, che ci dica se una determinata soluzione per un problema è ottima. Privilegiando l'efficienza a costo dell'ottimalità, si utilizzano *algoritmi di approssimazione* tali per cui la soluzione restituita non è per forza ottimale, ma ha costo C tale che:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

12.1 Copertura dei vertici

Dato un grafo non orientato $G = (V, E)$, una copertura dei vertici è un sottoinsieme $V' \subseteq V$ tale che, se $(u, v) \in E$, allora $u \in V' \vee v \in V'$. La dimensione di una copertura V' è il numero di vertici che contiene. Il problema della copertura dei vertici consiste nel trovare V' ottima (minimizzarla). Decidere se una certa copertura è ottima è un problema NP-completo.

Algoritmo 34 APPROX-COVER(G)

```
 $C \leftarrow$  insieme vuoto  
 $E' \leftarrow E$   
while  $E' \neq \emptyset$  do  
    sia  $(u, v)$  un arco in  $E'$   
     $C \leftarrow C \cup \{u, v\}$   
    rimuovi da  $E'$  ogni arco incidente in  $u$  o  $v$   
end while  
return  $C$ 
```

La complessità è $\mathcal{O}(n + m)$ se G è rappresentato con liste di adiacenza.

Fattore di approssimazione

APPROX-COVER ha un fattore di approssimazione di 2; si dice che è *2-approssimato*.

Dimostrazione. Sia A l'insieme di archi (u, v) scelti all'inizio di ogni ciclo. Per coprire gli archi in A , una copertura ottima deve contenere almeno un vertice di ciascun arco in A . Due archi in A non possono avere un vertice in comune, perchè una volta scelto (u, v) tutti gli archi incidenti ad u o v sono rimossi da E' . Pertanto, non ci sono due archi in A coperti dallo stesso vertice e $|A|$ è un

limite inferiore per la dimensione di una copertura ottima C^* . Ma $|C| = 2|A|$, poichè nel risultato metto sia u che v , quindi

$$|C| = 2|A| \leq 2|C^*|$$

□

12.2 Problema del commesso viaggiatore

Dato un grafo pesato, completo e non orientato bisogna trovare un *ciclo Hamiltoniano* di costo minimo. Il problema di capire se una soluzione è ottima è esso stesso NP-completo, di conseguenza vengono usate tecniche di approssimazione,

Il peso di un MAR è un limite inferiore per il peso di un circuito Hamiltoniano. Circumnavigando il MAR si ottiene un ciclo (con peso doppio rispetto al MAR) che però non è Hamiltoniano perchè passa dallo stesso nodo due volte. Per trasformarlo in Hamiltoniano, si possono tagliare le ripetizioni dei nodi, sfruttando la proprietà della disuguaglianza triangolare. È possibile utilizzare una DFS per passare subito dall'albero al circuito. L'algoritmo è 2-approssimato.

Algoritmo 35 APPROX-TSP(G, W, r)

scegli un vertice r casualmente

$A \leftarrow \text{PRIM}(G, W, r)$

$ord \leftarrow \text{DFS}(A, r)$

return ord

12.3 Ricerca locale

Una tecnica generale per trovare soluzioni buone è quella della *ricerca locale*. In un problem di ottimizzazione:

1. si parte da una soluzione ammissibile x
2. si calcolano tutte le soluzioni in un suo vicinato $N(x)$
3. se esiste una soluzione y appartenente a $N(x)$ migliore di x , allora $x \leftarrow y$ e si riparte dal punto 2
4. altrimenti, x è un minimo (o massimo) locale e ci possiamo fermare

Per utilizzare la ricerca locale dobbiamo essere in grado di generare almeno una soluzione e di definire un criterio per generare in maniera efficiente un vicinato $N(x)$.

Ricerca locale per TSP

Tecnica dei *k-scambi*:

1. si genera un ciclo Hamiltoniano casuale
2. se ne cancellano k archi non adiacenti e se ne aggiungono altri k in modo da ricreare il ciclo
3. se così facendo si crea un ciclo di peso minore, si ricomincia l'iterazione dal punto 2
4. altrimenti, si è trovata una soluzione minima localmente