Analisi e Progetto di Algoritmi

Fabio Ferrario

Gennaio 2022

Indice

Capitolo 1

Algoritmi su Grafi

1.1 Riassunto sui grafi

Indico un paio di definizioni:

- Definizione di Grafo: Un grafo è un insieme di elementi detti nodi o vertici che possono essere collegati fra loro da linee chiamate archi o lati o spigoli. Più formalmente, si dice grafo una coppia ordinata $G = (V, E)$ di insiemi, con V insieme dei nodi ed E insieme degli archi, tali che gli elementi di E siano coppie di elementi di V

- Grafo semplice: Grafo non orientato che non comprende cappi e archi multipli

- Grafo completo: è un grafo semplice nel quale ogni vertice è collegato a tutti gli altri vertici, quindi $n_{archi} = n_{vertici} * (n_{vertici} - 1)/2$

- Albero: Grafo non orientato G connesso tc: G è aciclico $\vee$ $|E| = |V| - 1$

- Albero Binario: Albero nel quale ogni nodo ha $\{0, 1, 2\}$ figli

- DAG: Grafo diretto (orientato) senza cicli, quindi aciclico. Un grafo diretto può dirsi aciclico se una visita in profondità NON presenta archi all'indietro.

1.2 Algoritmi di Visita

Esistono due tipi di algoritmi di visita dei grafi, in ampiezza (BFS) e in profondità (DFS)

Colore dei nodi $col[u]$ Sia BFS che DFS per funzionare assegnano un colore ad ogni vertice per capire se è già stato scoperto o no

- White: Vertice non ancora scoperto
- Gray: Vertice scoperto ma la cui lista di adiacenza non è ancora stata scandita del tutto
- Black: Vertice scoperto e di cui ho scandito per intero la lista di adiacenza

Predecessore del nodo $\pi[u]$ Il predecessore di u è il nodo che mi ha permesso di scoprirlo, quindi quello da cui sono "passato" nella mia ricerca per scoprire u

Tempo di scoperta del nodo $d[u]$ Il tempo di scoperta di u indica quanti "passaggi", quindi quanti nodi ho scoperto prima, mi ci sono voluti per arrivare a u

1.2.1 Algoritmo BFS

BFS (Breadth-First search) è un algoritmo di visita di un grafo in ampiezza. BFS scopre tutti i vertici raggiungibili partendo dal vertice sorgente, ma soltanto della componente connessa alla sorgente.
La scoperta avviene in ampiezza, ovvero parte da tutti i vertici a distanza 1 dalla sorgente, poi 2 e così via.
Alla fine dell'esecuzione di BFS, tutti i vertici della compoente connessa a cui appartiene il vertice sorgente avranno colore NERO.

Pseudocodice Algoritmo BFS

```
BFS(G, s)
    for ogni v ∈ V\{s}
        col[v] = White
        d[v] = ∞
        π[v] = NIL
    col[s] = Gray
    d[s] = 0
    π[s] = NIL
    ENQUEUE(Q, s)
    While Q≠∅
        u = DEQUEUE(Q)
```

```
for ogni v ∈ adj[u]
    if col[v]==White
        col[v] = Grigio
        d[v] = d[u] +1
        π[v] = u
        ENQUEUE(Q, v)
col[u] = Nero
```

Spiegazione Codice BFS inizializza tutti i nodi del grafo (tranne s) in modo da renderli "elaborabili". In seguito crea una coda Q che andrà a contenere tutti i nodi grigi, quindi quelli di cui va ancora completata la lista di adiacenza, dove inserisce s, che è il primo nodo Grigio.
Nel ciclo while l'algoritmo prende il primo nodo da Q e va a scoprire tutti i nodi bianchi (quindi non ancora elaborati) nella sua lista di adiacenza, inserendoli di volta in volta in Q. Quando la lista di adiacenza è stata scandita per intero, vuol dire che il nodo è stato scoperto del tutto e quindi diventa nero. Il ciclo ricomincia finchè Q non sarà vuota.

Sottografo dei predecessori Oltre a segnare la distanza la visita in ampiezza costruisce un albero BF (Albero di ricerca in ampiezza), che alla radice ha il vertice sorgente s. Quando un vertice v Quando durante l'ispezione della lista di adiacenza di un vertice u viene scoperto un vertice vianco v, il vertice v e l'arco (u, v) che vengono aggiunti all'albero. Il vertice u viene detto padre di v.

1.2.2 Algoritmo DFS

DFS (Depth-First search) è un algoritmo di visita di un grafo in profondità. Alla fine dell'esecuzione dell'algoritmo siamo in grado di determinare quante sono le componenti connesse del grafo ed a quale componente connessa appartiene ogni nodo.

Classificazione degli archi Dato un grafo orientato, DFS differenzia ogni arco (u, v) in 4 modi diversi

- Arco dell'albero: $col[v] = BIANCO$, quindi è la prima volta che visitamo v
- Arco all'indietro: $col[v] = GRIGIO$, quindi v è antenato di u

- Arco in avanti: $col[v] = NERO \wedge d[u] < d[v]$, quindi v è stato scoperto dopo u
- Arco di attraversamento: $col[v] = NERO \wedge d[u] > d[v]$, quindi v è stato scoperto prima di u

In un grafo NON ORIENTATO esistono soltanto gli archi dell'albero e gli archi all'indietro.

Pseudocodice Algoritmo DFS

```
DFS(G)
    for ogni u ∈ V
        col[u] = White
        π[u] = NIL
    time = 0
    for ogni u ∈ V
        if col[u] == White
            DFS–VISIT(u)

DFS–VISIT(G,u)
col[u] = Gray
time++
d[u] = time
for ogni w ∈ Adj[u]
    if color[w] == White
        π[w] = u
        DFS–VISIT(G,w)
col[u] = Black
time ++
f[u] = time
```

Spiegazione pseudocodice DFS comincia inizializzando il grafo come BFS ma senza impostare d[]. Poi fa un ciclo su ogni nodo del grafo, chiamato DFS-VISIT su tutti i nodi bianchi che trova. DFS-VISIT(G,u) visita tutti i nodi adiacenti a u, operando ricorsivamente su ogni nodo bianco che trova. Alla fine dell'esecuzione DFS-VISIT mette nero il nodo in esame e procede con il primo nodo bianco che trova nell'albero.

Sottografo dei predecessori Data una visita in profondità, il sottografo dei predecessori può essere formato da pià alberi, poichè la visita può essere

ripetuta da più sorgenti. Quindi forma una foresta DF, composta da vari alberi DF.

1.3 Cammini minimi

Sia G=(V,E) un grafo orientato con costi w sugli archi, il costo di un cammino p=<$v_1, v_2, ...v_k$>è dato dalla somma del peso di tutti i vertici di quel cammino

Cammino minimo tra una coppia di vertici x e y è un cammino di costo minore o uguale a quello di ogni altro cammino tra gli stessi vertici

Sottostruttura ottima ogni sottocammino di un cammino minimo è anch'esso minimo

Albero dei cammini minimi I cammini minimi da un vertice s a tutti gli altri vertici del grafo possono essere rappresentati tramite un albero radicato in s, detto albero dei cammini minimi

1.4 Algoritmo di Floyd-Warshall

L'algoritmo di Floyd-Warshall calcola il cammino minimo da i a j per ogni coppia di vertici (i, j) del grafo (pesato e orientato) su cui viene eseguito

Funzionamento L'idea alla base di questo algoritmo è un processo iterativo che, scorrendo tutti i nodi, ad ogni passo h si ha (data una matrice D) nella posizione [i,j] la distanza - pesata - minima dal nodo di indice i a quello j attraversando solo nodi di indice minore o uguale a h.
Quindi D^h equivale alla matrice che contiene i cammini minimi utilizzando come nodi intermedi al massimo i nodi di indice h.
Se non vi è collegamento tra due nodi allora nella cella corrispondente c'è infinito. Ovviamente alla fine (con h = numero di nodi) leggendo la matrice si ricava la distanza minima fra i vari nodi del grafo. L'algoritmo di Floyd-Warshall è un algoritmo di programmazione dinamica bottom-usepackage

Equazione di Ricorrenza

$$d_{i,j}^{(h)} = \begin{cases} W_{ij}, & \text{if } h = 0 \\ min\{d_{ij}^{(h-1)}, d_{ih}^{(h-1)} + d_{jh}^{(h-1)}\} & \text{if } h > 0 \end{cases}$$

Pseudocodice Algoritmo di Floyd-Warshall

```
FLOYD–WARSHALL (G, W)
    D^(0) = W
    for h = 1 to n
        for i = 1 to n
            for j = 1 to n
                d_ij^(h) min{d_ij^(h-1), d_ih^(h-1) + d_jh^(h-1)}
```

Spiegazione Codice L'algoritmo inizia impostando a zero tutti i cammini minimi di un nodo con se stesso. Procede poi con tre cicli for, il cui più esterno è h e il più interno j, in cui confronta i cammini minimi di ogni nodo con ogni altro nodo aumentando di volta in volta il (possibile) numero di nodi intermedi (h)

Tempo di esecuzione

$$O(|V|^2)$$

1.4.1 Variante: Cammini minimi $\leq L$

Dato un grafo orientato e senza cappi (V, E, W) e dato un itnero L>0 calcolare $\forall (i,j) \in V^2$ il peso di un cammino minimo da i a j di lunghezza $\leq L$

Variabili introdotte $D^{(k,l)} = (d_{ij}^{(k,l)})$
dove $d_{ij}^{(k,l)}$ è il peso del cammino minimo da i a j con vertici intermedi $\in \{1, ..., k\}$ di lunghezza $\leq l$

Caso base (k,l) con k=0

$$d_{ij}^{(0,l)} = \begin{cases} 0 & \text{if } i = j \\ w_{ij} & \text{if } i \neq j \wedge (i,j) \in E \\ \infty & \text{altrimenti} \end{cases}$$

Spiegazione caso base Se i = j allora la distanza è 0, siccome il cammino deve essere di lunghezza ≤l un cammino di lunghezza 0 è accettato
Se $i \neq j \wedge (i,j) \in E$ allora è w_{ij}, perchè siccome cè un solo cammino sarà sempre di lunghezza 1, che è minore o uguale ad ogni l
Infinito altrimenti, perchè non c'è un cammino che collega i a j

Passo ricorsivo (k, l) con $k > 0$

$$d_{(ij)}^{(k,l)} = \begin{cases} min\{d_{ij}^{(k-1,l)}, d_{(ik)}^{(k-1,l_1)} + d_{(kj)}^{(k-1,l_2)}\} & \text{if } l > 1 \text{ e } l_1, l_2 \in \{1, ..., l\}, l_1 + l_2 \leq l \\ min\{d_{ij}^{(k-1,l)}, \infty\} & \text{if } l = 1 \end{cases}$$

Spiegazione Passo ricorsivo Se $l > 1$ vuol dire che ci può essere un vertice intermedio k tra i e j, quindi bisogna trovare il minimo tra la distanza a k-1 e la somma di due percorsi di lunghezza minore di l tra due percorsi (ovviamente minori di k) che utilizzando k come intermedio e la cui somma non superi l.

Invece se k non fa parte del cammino minimo, si sceglie il minore tra un eventuale percorso tra i e j che non include k oppure infinito.

```
CAMMINIMINIMI_MINORE_DI_L(V,E,W,L)
for l=1 to L //Caso base puro (k=0)
    for i=1 to n
        for j=1 to n
            if i==j
                d_ij^(0,l) = 0
            elseif (i,j) ∈ E
                d_ij^(0,l) = w_ij
            else
                d_ij^(0,l) = ∞
for k=1 to n //Caso passo
    for l=1 to L
        for ogni i
            for ogni j
                if l==1
                    d_ij^(k,l) = min{d_ij^(k-1,l), ∞}
                else
                    for l_1 = 0 to L
                        for l_2 = 0 to L
                            if l_1 + l_2 ≤ L
                                d_ij^(k,l) = min{d_ij^(k-1,l), d_ik^(k-1,l_1) + d_jk^(k-1,l_2)}
```

1.5 Cammini minimi da sorgente unica

Gli algoritmi di Dijkstra e di Bellman-Ford risolvono il problema dei cammini minimi da sorgente unica. Quindi vengono usati quando vogliamo trovare un cammino minimo che va da un dato vertice sorgente $s \in V$ a ciascun vertice $v \in V$ in un grafo orientato pesato $G = (V, E)$

Differenze Dijkstra funziona soltanto se tutti i pesi degli archi sono NON NEGATIVI, mentre Bellman-Ford non ha bisogno di questa premessa

Funzionamento comune Come tutti gli algoritmi per cammini minimi entrambi si basano sulla proprietà della Sottostruttura ottima di un cammino. In questi algoritmi vengono assegnati due attributi per ogni vertice del grafo:

- $\pi(v)$ Che indica il predecessore di v nel cammino minimo
- $d(v)$ Che indica la distanza di v dal nodo sorgente s

Inoltre questi algoritmi hanno bisogno di due funzioni d'appoggio, INITIALIZE e RELAX

Inizializzazione del grafo Per gli algoritmi di questo tipo viene spesso utilizzata una funzione INITIALIZE, che imposta le distanze e i "padri" di ogni nodo rendendo s la nostra sorgente (quindi a distanza zero)

```
INITIALIZE(G, s)
    for ogni v ∈ V
        v.d = ∞
        v.π = NIL
    s.d = 0
```

Tecnica del rilassamento La tecnica del rilassamento di un $arco(u, v)$ consiste nel verificare se, passando per u, è possibile migliorare il cammino minimo per v precedentemente trovato. Quindi partendo da stime per eccesso delle distanze le decrementiamo progressivamente fino a renderle esatte

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

In sostanza, se la distanza del vertice v è maggiore della distanza di u più il peso dell'arco che va da u a v, allora sostituisci la distanza di v con $u.d + w(u, v)$ e imposta u come padre di v

1.5.1 Algoritmo di Dijkstra

Dijkstra ritorna in output l'insieme S contenente tutti i cammini minimi per ogni nodo del grafo In Dijkstra il rilassamento viene eseguito esattamente una volta per arco.

Attenzione Dijkstra funziona solo se w $\geq 0 \; \forall$ w $\in$ W

Pseudocodice Algoritmo di Dijkstra

```
DIJKSTRA(G, w, s)
    INITIALIZE(G, s)
    S = φ
    Q = G.V
    While Q ≠ φ
        u = extract–min(Q)
        S = S ∪u
        for ogni vertice v ∈ Adj[u]
            RELAX(u, v, w)
```

Spiegazione codice Q è una coda che contiene tutti i vertici del grafo ed S è l'insieme delle soluzioni. Viene estratto il vertice con distanza minore dalla sorgente (al primo giro sarà sempre s dato che è a distanza 0) e viene aggiunto all'insieme delle soluzioni. Viene poi fatto il rilassamento per ogni vertice adiacente a quello aggiunto alla soluzione, aggiornandone la stima della distanza dalla sorgente e il predecessore. viene ripetuto per ogni nodo rimanente nella coda Q.

Greedy Dijkstra segue l'approccio Greedy, quindi sceglie sempre il vertice più "leggero" o "vicino" in V\S da aggiungere all'insieme S. Vi è un'analogia con il criterio per l'ordinamento negli algoritmi Greedy

Tempo di esecuzione

- $O(E + |V| * log|V|)$ Se utiliziamo lo Heap di Fibonacci per estrarre velocemente il nodo a distanza minore nella coda
- $O(|V|^2)$ Altrimenti (implementazione naive)

1.5.2 Algoritmo di Bellman-Ford

Attenzione Bellman-Ford funziona anche se i pesi degli archi sono negativi

Funzionamento Bellman-Ford ritorna in Output un valore booleano che indica se esiste oppure no un ciclo di peso negativo che è raggiungibile dalla sorgente. Se tale ciclo non esiste, l'algoritmo fornisce i cammini minimi e i loro pesi. L'algoritmo restituisce true se e solo se il grafo non contiene cicli di peso negativo che sono raggiungibili dalla sorgente.
In Bellman-Ford il rilassamento viene eseguito $|V| - 1$ volte per arco indipendentemente dalla morfologia del grafo. Bellman-Ford NON segue un approccio Greedy.

Pseudocodice algoritmo di Bellman-Ford

```
BELLMAN-FORD(G, w, s)
    INITIALIZE(G, s)
    for i = 1 to |V|-1
        for ogni (u,v) ∈ E
            RELAX(u, v, w)
    for ogni (u,v) ∈ E
        if v.d > u.d + w(u, v)
            return FALSE
    RETURN TRUE
```

Spiegazione codice Dopo aver inizializzato il grafo, Bellman-Ford procede rilassando ogni arco $|V| - 1$ volte, Se il grafo ha N nodi è certo che dopo N-1 giri tutti i nodi hanno a loro assegnato il costo minimo per essere raggiunti dal nodo sorgente. L'ultimo ciclo controlla se ci sono cicli di peso negativo, in tal caso ritorna FALSE

Tempo di esecuzione

$$O(|V| * |E|)$$

Capitolo 2

Preparazione dell'Esame

2.1 Esercizi in generale

2.1.1 Varianti di FW

Un possibile tipo di esercizio d'esame è problema in cui is chiede di creare un algoritmo di programmazione dinamica che, dato un grafo pesato e una funzione che associa un numero o un colore a ogni arco o vertice, stabilisce il peso di/se seiste un cammino minimo da i a j che rispetti alcuni vincoli.

La trafila e sostanzialmente sempre la stessa , bisogna trovare un sottoproblema k-esimo e capire se le informazioni date da i vertici (i, k) e (k, j) sono sufficienti a risolvere il problema. in caso contrario bisogna definire un problema ausiliario.

Variabili associate del sottoproblema (quindi quelle introdotto) sono sempre: il matricione D^k in cui d_{ij}^k è il peso/l'esistenza del cammino da i a j i cui vertici intermedi appartengono a $\{1, ..., k\}$

soluzione del sottoproblema Se le informazioni sono sufficienti, si formano le due equazioni di ricorrenza in cui si calcola d_{ij}^k, per il caso base e per il Passo Ricorsivo.
Il passo ricorsivo (k¿0) è sempre diviso in due casi ipotetici (sibilla cumana), il caso in cui k NON fa parte del cammino minimo, in cui il valore di d_{ij}^k diventerà il valore del cammino ij escludendo k come cammino intermedio, quindi k-1.
Oppure il caso in cui k fa parte del cammino minimo, in cui il valore di d_{ij}^k diventerà la somma (oppure l'AND) dei valori dei cammini ik e kj (tutti e

due escludendo k come cammino intermedio)
L'equazione di ricorrenza diventerà quindi il minimo dei due casi (oppure un OR nel caso in cui si voglia verificare l'esistenza).
Il caso base invece è dove i calcoli veri vengono fatti. in genere il valore di d_{ij}^k ha tre possibilità: se i=j assumerà 0 o TRUE, perchè se i e j coincidono vuol dire che non ci sono archi per passare da uno all'altro. Se i e j non coincidono, il loro cammino è compreso in E e sono rispettate le condizioni del problema allora d_{ij}^k diventerà il peso del cammino (oppure TRUE). Diventa sempre ∞ o FALSE altrimenti.

Problema ausiliario Nel caso in cui non sia possibile calcolare il risultato con le informazioni date è necessario introdurre un Problema ausiliario. Questo problema generalmente è lo stesso iniziale ma aggiunge un controllo aggiuntivo, per esempio sul primo arco uscente e l'ultimo entrante del cammino. Una volta introdotto il problema ausiliario si procede come se fosse un problema normale, quindi si introduce il sottoproblema, le variabili e le equazioni di ricorrenza. Con le equazioni di ricorrenza però vengono aggiunte delle condizioni particolari che dipendono caso per caso (guarda esercizi).

Solzuone del problema La soluzone del problema nel caso in cui non ci sia un problema ausiliario è semplicemente formata da i valori nel matricione introdotto dal sottoproblema (D), se invece è presente un problema ausiliario, allora la soluzione diventa il minimo (o OR) di tutte le soluzioni del Problema ausiliario

2.2 Esattamente 3 vertici BLU

Es 2 esame 2017
Grafo G dove ad ogni vertice è associato un colore R,V,B. Stabilire per ogni coppia di vertici se esiste un cammino con esattamente 3 vertici blu

Variabili Introdotte

$D^{n,b}$ Matrice $|V| \cdot |V|$
$d_{ij}^{(k,b)}$ = True se esiste un cammino da i a j con esattamente b vertici BLU che usa al più 1,...,k vertici intermedi.
con $b = \{0, ..., 3\}$ e $k = \{0, ..., |V|\}$

Caso Base

k=0
Tre casi, con $b \in \{0, 1, 2\}$ perchè $b < k + 2$ siccome con 0 vertici intermedi il cammino ij ha al più due vertici

$$d_{ij}^{(0,0)} = \begin{cases} TRUE & i = j \wedge col(i) \neq B \wedge col(j) \neq B \\ TRUE & i \neq j \wedge (i,j) \in E \wedge col(i) \neq B \wedge col(j) \neq B \\ FALSE & \text{Altrimenti} \end{cases}$$

$$d_{ij}^{(0,1)} = \begin{cases} TRUE & i = j \wedge col(i) = col(j) = B \\ TRUE & i \neq j \wedge (i,j) \in E \wedge col(i) = BXORcol(j) = B \\ FALSE & \text{Altrimenti} \end{cases}$$

$$d_{ij}^{(0,2)} = \begin{cases} TRUE & i \neq j \wedge (i,j) \in E \wedge col(i) = col(j) = B \\ FALSE & \text{Altrimenti} \end{cases}$$

Passo ricorsivo

Se $k \notin cammino$ allora

$$d_{ij}^{(k,b)} = d_{ij}^{(k-1,b)}$$

che per comodità chiameremo $e1$

Se $k \in cammino$ allora dobbiamo distinguere due casi:

$col(k) \neq B$

$$d_{ij}^{(k,b)} = d_{ik}^{(k-1,b_1)} \wedge d_{kj}^{(k-1,b_2)}$$

in cui $b_1 + b_2 == b$, che per comodità chiameremo $e2_b$

$col(k) \neq B$

$$d_{ij}^{(k,b)} = d_{ik}^{(k-1,b_1)} \wedge d_{kj}^{(k-1,b_2)}$$

in cui $b_1 + b_2 == b + 1$, che per comodità chiameremo $e2_a$

Quindi l'equazione del passo ricorsivo è:

$$d_{ij}^{(k,b)} = \begin{cases} e1 \vee e2_b & k = B \\ e1 \vee e2_a & k \neq B \end{cases}$$

Soluzione del problema

La soluzione del Problema è contenuta in tutti i valor di $D^{|V|,3}$

2.3 Esercizio cammino minimo vertici con segno alternato

Esercizio 1 esame 12/09/2016
Sia A l'insieme dei numeri interi escluso 0.
Dato un grafo non orientato (V,E,f) in cui ad ogni vertice è associato un numero intero diverso da zero (mediante la funzione $f : V \to A$), mediante la tecnica della programmazione dinamica si vuole stabilire per ogni coppia di vertici (i, j), se esiste un cammino da i a j avente vertici che danno luogo ad una alternanza del segno dei numeri ad esso associati. RISPONDERE PER PUNTI alle seguenti richieste:

1. Esplicitare e definire le variabili che servono per risolvere il problema
2. Scrivere l'equazione di ricorrenza per il CASO BASE, giustificando perché è fatta in quel modo
3. scrivere la/le equazione/i di ricorrena per il PASSO RICORSIVO, giustificando perché è/sono fatta/e in quel modo
4. Scrivere qualè la soluzione del problema, espressa rispetto alle variabili introdotte

Variabili introdotte

D^k matrice $|V| \cdot |V|$
$d^k_{i,j}$ True sse esiste un cammino da i a j che da luogo ad una alternanza di segno usando $1, ..., k$ vertici intermedi

Caso Base

Non ci sono vertici intermedi $k = 0$

$$d^0_{i,j} = \begin{cases} TRUE & \text{se } i = j \\ TRUE & \text{se } i \neq j \wedge (i,j) \in E \wedge f(i) \cdot f(j) < 0 \\ FALSE & \text{altrimenti} \end{cases}$$

Passo Ricorsivo

$k > 0$
due casi:

- $k \notin$ cammino $\rightarrow d_{ij}^{k} = d_{ij}^{k-1}$
- $k \in$ cammino $\rightarrow d_{ij}^{k} = d_{ik}^{k-1} \wedge d_{kj}^{k-1}$

Quindi, nel passo ricorsivo con $k > 0$:

$$d_{ij}^{k} = d_{ij}^{k-1} \vee (d_{ik}^{k-1} \wedge d_{kj}^{k-1})$$

Soluzione del problema

La soluzione del problema è costituita da tutti i valori contenuti in $D^{|V|}$

2.4 Esercizio Arco RED mai seguito da arco BLUE

Dato un grafo G(V,E,W) orientato e senza cappi in cui ad ogni arco è associato un colore tramite la funzione $col : E \rightarrow \{red, blue\}$ calcolare $\forall (i,j) \in V^2$ il peso di un cammino minimo da i a j nella quale un arco *red* non è mai seguito da un arco *blue*

Definiamo il Problema Ausiliario P' Dato $G \forall (a,b) \in C^2$ calcolare $\forall (i,j) \in V^2$ il peso di un cammino minimo da i a j con colore del I arco pari ad a e dell'iltimo a b

Sottoproblema k**esimo di** P' con $k \in \{0, ..., n\}$ $\forall (a,b) \in C^2$ calcolare $\forall (i,j) \in V^2$ il peso di un cammino minimo da i a j con colore del I arco pari ad a e dell'iltimo a b e con vertici intermedi $\in \{0, ..., k\}$

Introduco la variabile $D*(k,a,b)$ $\forall (i,j) \in V^2$, $d^{(k,a,b)_{ij}}$ è il peso di un cammino minimo da i a j con colore del I arco pari ad a e dell'iltimo a b e con vertici intermedi $\in \{0, ..., k\}$
Introduco la variabile $D*(k,a,b)$

Caso Base sottoproblema di P' $k = 0$

$$d_{ij}^{(0,a,b)} \begin{cases} \infty & \text{se } i = j \\ w_{ij} & \text{se } i \neq j \wedge (i,j) \in E \wedge col(i,j) = a = b \\ \infty & \text{altrimenti} \end{cases}$$

Passo ricorsivo sottoproblema di P' $k > 0$
Abbiamo due casi
se $k \notin cammino$ allora $d_{ij}^{(k,a,b)} = d_{ij}^{(k-1,a,b)}$
se $k \in cammino$ allora

grafo: i [a]- - -[c] k [d]- - -[b] j con $(c,d) \neq (red, blue)$ tradotto: Se k appartiene al cammino minimo, allora il colore dell'ultimo arco entrante in k deve essere diverso da red, oppure il colore del primo arco uscente da k deve essere diverso da blue. quindi: $d_{ij}^{(k,a,b)} = d_{ik}^{(k-1,a,c)} + d_{kj}^{(k-1,d,b}$ con $c, d \in C^2$ tc $c \neq red \vee b \neq blue$

Quindi l'equazione di ricorrenza del passo ricorsivo è:
$d_{ij}^{(k,a,b)} = min\{d_{ij}^{(k-1,a,b)}, d_{ik}^{(k-1,a,c} + d_{kj}^{(k-1,d,b}\}$ con $c, d \in C^2$ tc $c \neq red \vee b \neq blue$

Soluzione Problema Ausiliario P' è formata dalle matrici:
$D^{(n,red,red)}$ $D^{(n,red,blu)}$ $D^{(n,blu,red)}$ $D^{(n,blu,blu)}$
Ovvero tutte le combinazioni possibili di cammini minimi con gli archi a e b = red or blu

Solzione Problema di partenza PB

$$d_{ij}^{prob} = \begin{cases} 0 & \text{se } i = j \\ min\{d_{ij}^{(n,a,b)} \forall (a,b) \in C^2\} & \text{altrimenti} \end{cases}$$

2.5 Esercizio k alberi e almeno h vertici

Esercizio 2 esame 12/09/2016
Scrivere un algoritmo che, dati un grafo non orientato e due interi positivi $h > 0$ e $k > 0$, stabilisce se ENTRAMBE le seguenti condizioni sono verificate

- Esattamente k componenti connesse del grafo sono alberi
- Ogni componente connessa del grafo ha almeno h vertici

Spiegazione

Per creare questo algoritmo occorre modificare DFS:

- Per capire quanti alberi ci sono, inizializzo un boolean TREE a True. In visit controllo gli archi all'indietro e metto TREE = False se ne trovo. Al ritorno in DFS, se TREE è True incremento un contatore che alla fine di DFS controllerò che equivalga a k
- Per capire il numero di Vertici in una CC, in VISIT incremento un contatore ricorsivamente ogni volta che incontro un nodo white. Una volta tornato in DFS controllo che la CC abbia almeno h vertici.

Pseudocodice

```
DFS–CONDITIONS(G,h,k)
    for ogni u ∈ V
        col[u] = WHITE
        π[u] = NIL
    tree = True      #variabile globale
    n_trees = 0
    for ogni u ∈ V
        if col[u] == White
            n_vertici = DFS–VISIT–CONDITIONS(u)
            if n_vertici < h
                RETURN False
            if tree == True
                n_trees = n_trees + 1
            else
                tree = true
    if n_trees == k
        RETURN True
    else
        RETURN False

DFS–VISIT–CONDITIONS(u)
    col[u] = Gray
    n_vertici = 0
    for ogni w ∈ Adj[u]\π[u]
        if col[w]==White
            π[w] = u
            n_vertici = n_vertici + DFS–VISIT–CONDITIONS(w)
```

```
            else if col[w] == Gray and tree = True
                tree = False
        col[u] = Black
        return n_vertici + 1
```

2.6 Esercizio k grafi completi e h vertici

Dati due interi h,k¿0 e un grafo non orientato in cui ad ogni vertice è associato un simbolo ($ e *) stabilisci se: ogni CC del grafo ha almeno h vertici a cui è associato il simbolo $ e ci sono al più k componenti connesse del grafo che prese singolarmente sono grafi completi

spiegazione

- h vertici per cc..., ogni volta che trovo un vertice white aumento un counter se è associato al dollaro, e alla fine controllo se è ¿ h
- k cc sono.... un grafo è completo se ci sono $n_{vert} \cdot (n_{vert} - 1)/2$ archi. per contare gli archi si incrementa un counter ogni volta che si trova un vertice nella lista di adiacenza.

Pseudocodice

```
DFS_CUSTOM(G,h,k)
    for ogni u ∈ V
        col[u] = WHITE
        π[u] = NIL
    n_completi = 0
    n_archi = 0
    for ogni u ∈ V
        if col[u] == WHITE
            n_dollari = DFS_VISIT_CUSTOM(u)
            if n_dollari < h
                RETURN False
            if n_archi = n_vertici · (n_vertici − 1)/2
                n_completi = n_completi + 1
            n_archi = 0
    if n_completi > k
        RETURN False
    RETURN True
```

```
DFS_VISIT_CUSTOM(u)
    col[u] = GRAY
    if f(u) == "dollar"
        n_dollar = 1
    else
        n_dollar = 0
    for ogni w ∈ Adj[u]\π[u]
        n_archi = n_archi + 1
        if col[w] == WHITE
            π[w] = u
            n_dollar = n_dollar + DFS_VISIT_CUSTOM(w)
    RETURN n_dollar
```

2.7 Esercizio Generico conta vertici ∀ CC

Esercizio generico (inventato da me), scrivi un algoritmo che conta i vertici in ogni componente connessa di un grafo

Spiegazione

è una modifica di DFS in cui DFS-VISIT ritorna il numero di vertici che ha trovato nella componente connessa del nodo in esame. Quindi dopo ongi chiamata di DFS-VISIT, DFS si ritrova con il numero dei vertici e può farci quello che vuole. DFS-VISIT funziona come di norma (senza usare i padri che non servono) ma inizializza un contatore (che poi ritornerà) e ogni volta che fa una chiamata ricorsiva aggiorna il valore.

Pseudocodice

```
DFS_COUNT(G)
    for ogni u ∈ V
        col[u] = WHITE
    for ogni u ∈ V
        if col[u] == WHITE
            num_in_cc = DFS_VISIT_COUNT(u)
            //Qui fai quello che vuoi con num_in_cc

DFS_VISIT_COUNT(u)
    col[u] = GRAY
    count = 0
```

```
for ogni v ∈ adj[u]
    if col[v] == WHITE
        count = count + DFS_VISIT_COUNT(v)
col[u] = BLACK
RETURN count + 1
```

2.8 Esercizio conta CC di un grafo

Scrivere un algoritmo che determina il numero di componenti connesse di un grafo G = (V, E) non orientato.

Spiegazione

Ogni volta che DFS invoca DFS-VISIT significa che ha trovato una componente connessa, quindi basta mettere un contatore che incrementa ogni volta che si chiama DFS-VISIT da DFS

Pseudocodice

```
DFS_COUNT_CC(G)
    for ogni u ∈ V
        col[u] = WHITE
    count = 0
    for ogni u ∈ V
        if col[u] == WHITE
            count = count + 1
            DFS_VISIT(u)
    RETURN count
```

2.9 Esercizio Controlla se G è Aciclico

Modificare l'algoritmo DFS di visita di un grafo orientato G in maniera tale che stabilisca se G è acicilico, ossia se non contiene cicli.

Spiegazione

Un grafo è aciclico se non contiene nessun arco all'indietro (btw se un grafo non orientato è aciclico allora è un albero).

Quindi bisogna semplicemente iniziallizzare un Boolean "acyclic" a TRUE e farlo diventare FALSE qual'ora si incontrasse un arco all'indietro

Pseudocodice

```
DFS–ACYCLIC(G)
    for ogni u ∈ V
        col[u] = WHITE
        π[u] = NIL
    time = 0
    acyclic = true
    for ogni u ∈ V
        if col[u] == WHITE
            DFS–VISIT–ACYCLIC(u)
    return acyclic

    DFS–VISIT–ACYCLIC(u)
        col[u] = GRAY
        time++
        d[u] = time
        for ogni w ∈ Adj[u]
            if color[w] == WHITE
                π[w] = u
                DFS–VISIT–ACYCLIC(w)
            else if col[w] == GRAY and acyclic == true
                acyclic = FALSE
        col[u] = black
        time ++
        f[u] = time
```

In caso di grafo NON ORIENTATO

Se il grafo non è orientato, l'algorimto è lo stesso ma nel controllo degli adiacenti di u in visit bisogna togliere il padre di u

2.10 Domande di Teoria

1 Siano $X =< x_1, \ldots, x_m >$ e $Y =< y_1, \ldots, y_n >$ due sequenze e sia $Z =< z_1, \ldots, z_k >$ una LCS di X e Y. Scrivere la proprieta' di sottostruttura

ottima di Z
(Teorema 15.1 pag 325)

1. Se $x_m = y_n$, allora $z_k = x_m = y_n$ è una LCS di X_{m-1} e Y_{n-1}
2. Se $x_m \neq y_n$, allora $z_k \neq x_m$ implica che Z è una LCS di X_{m-1} e Y
3. Se $x_m \neq y_n$, allora $z_k \neq y_n$ implica che Z è una LCS di X e Y_{n-1}

2 Scrivere le equazioni di ricorrenza per risolvere il problema della LCS di due sequenze, specificando bene il significato delle variabili coinvolte
("Fase 2: una soluzione ricorsiva" pag. 325)

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ and } x_i = y_j \\ max(c[i, j-1], c[i-1, j] & \text{se } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

3 Scrivere l'algoritmo che determina la lunghezza della LCS di due sequenze specificando il suo tempo di calcolo
("Fase 3: Calcolare la lunghezza di una LCS" pag.326) Tempo di esecuzione: $\theta(mn)$ perchè il calcolo di ogni posizione della tabella richiede 1.

```
LCS-LENGHT(X,Y)
    m = X.lenght
    n = Y.lenght
    Siano b[1..m,1..n] e c[0..m,0..m] due nuove tabelle
    for i = 1 to m
        c[i,0] = 0
    for j= 0 to n
        c[0,j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i == y_i
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = ↖
            elseif c[i - 1,j] ≥ c[i,j-1]
                c[i,j] = c[i-1,j]
                b[i,j] = ↑
            else
                c[i,j] = c[i,j-1]
                b[i,j] = ←
    return c e b
```

4 Definire qual e' il sottografo dei predecessori (o albero BFS) prodotto dalla visita BFS di un grafo G=¡V,E¿, specificando bene da quali vertici e quali archi e' composto.
("Alberi di visita in ampiezza" pag. 502)
Il sottografo dei predecessori G_π è un albero BF (o albero di visita in ampiezza) se V_π è formato da vertici raggiungibili da s e, per ogni v $\in V_\pi$, c'è un solo cammino semplice da s a v in G_π che è anche un cammino minimo da s a v in G. GLi archi in E_π sono detti archi dell'albero

5 Scrivere qual e' il tempo di calcolo dell'algoritmo BFS motivando BENE la risposta (fare riferimento allo pseudocodice).
("Analisi" pag 499)
Tempo di esecuzione: O(V + E)
Tempo delle operazioni con la coda O(V) e tempo per l'ispezione di ADJ è O(E)

6 Spiegare il significato dei colori assegnati ai vertici dall'algoritmo BFS. Alla fine dell'esecuzione dell'algoritmo BFS su un grafo G=¡V,E¿, quali colori assumono i vertici?
(Sez 22.2 pag 497)

7 Definire qual e' il sottografo dei predecessori (o foresta DF) prodotto dalla visita DFS di un grafo G=¡V,E¿, specificando bene da quali vertici e quali archi e' composto.
(Sez 22.3 pag 504)
Il sottografo dei predecesso o foresta DF è così definita $G\pi = (V, E\pi)$, dove: $E\pi = (\pi[v], v) : v \in V \wedge \pi[v] \neq NIL$

8 Descrivere la classificazione degli archi che la visita in profondita' produce a partire da un grafo G=¡V,E¿. Come si classifica un generico arco (u,v) del grafo?
("Classificazione degli archi" pag 509) Arco dell'albero, in avanti, indietro e di attraversamento

9 Scrivere qual e' il tempo di calcolo dell'algoritmo DFS motivando BENE la risposta (fare riferimento allo pseudocodice)
(Pag. 464-465)
Tempo di esecuzione DFS: $\theta(V + E)$ perchè: DFS-VISIT è chiamata una volta per ogni vertice (quando è bianco) e il ciclo in DFS-VISIT è chiamato una volta per ogni E (quindi ogni volta che c'è una adiacenza)

11 Dare la definizione di ordinamento topologico, specificando bene a che tipo di grafo si applica. Descrivere come si ottiene l'ordinamento topologico sfruttando l'algoritmo DFS.
(sez 22.4 pag 512)
Un ordinamento topologico di un DAG G è un ordinamento lineare di tutti i suoi vertici tale che, se G contiene un arco (u,v), allora u appare prima di v nell'ordinamento. TOPOLOGICAL-SORT(G): chiama DFS per calcolare i tempi di completamento v.f, poi completata l'ispezione inserisce il vertice in una lista concatenata che poi ritorna.

12 Descrivere la caratterizzazione della struttura di un cammino minimo p=¡ v1, . . . , vl¿ utilizzata dall'algoritmo di Floyd-Warshall.
(Paragrafo "La struttura di un cammino minimo" pag. 580)

13 Illustrare e motivare le equazioni di ricorrenza su cui si basa l'algoritmo di Floyd-Warshall, specificando bene il significato delle variabili coinvolte.

$$d_{ij}^{k} = \begin{cases} w_{ij} & k = 0 \\ min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1} & k > 0 \end{cases}$$

14 illustrare un metodo per costruire i cammini minimi nell'algoritmo di Floyd-Warshall.
Calcolo la matrice D dei pesi dei cammini minimi e poi costruisco la matrice dei predecessori della matrice D

15 Che cos'e' la chiusura transitiva di un grafo orientato? Descrivere un modo per calcolarla.
Dato un grafo orientato si vuole determinare se per ciascuna coppia di ij esuste un cammino da i a j. la chiusura transitiva è definita come il grafo G* = (V,E*) dove E* = (i,j): esiste un cammino dal vertice i al vertice j in G
un modo per calcolare la CT consiste nell'assegnare un peso 1 a ogni arco e nell'eseguire FW. se esiste un cammino da i a j si ha dij ¡ n, altrimenti infinito

16 escrivere come modificare le equzioni di ricorrenza dell'algoritmo di Floyd-Warshall per calcolare la chiusura transitiva di un grafo orientato.
k=0

$$t_{ij}^{0} = \begin{cases} 0 & i \neq j \wedge (i,j) \notin E \\ & i = j \vee (i,j) \in E \end{cases}$$

k¿0

$$t_{ij}^{k} = t_{ij}^{k-1} \vee (t_{ik}^{k-1} \wedge t_{kj}^{k-1})$$

17 Dare la definizione di 1) Sistema di Indipendenza, 2) Problema associato ad un coppia costituita da un sistema di indipendenza e da funzione peso definita sul sistema di indipendenza. Scrivere inoltre qual è l'algoritmo Greedy associato a tale coppia.
Sistema di indipendenza: Data la coppia ¡E,F¿ dove E è un insieme finito e F è una famiglia di sottoinsiemi di E, definiamo tale coppia sistema di indipendenza se vale al seguente proprietà:

$$\forall A \in FseB \subseteq A \implies B \in F$$

18 Definire cos'è un matroide, enunciare il Teorema di Rado
matroide: un sistema di independenza è detto matroide se:

$$\forall A, B \in Fe|A| = |B| + 1 \rightarrow \exists a \in A \backslash BtalecheB \cup \{a\} \in F$$

teorema di Rado: Dato un sitema di indipendenza ¡E,F¿ le seguenti proposizioni sono equivalenti:

- Per ogni funzione peso w : E -¿ R+, l'algoritmo greedy associato fornisce una soluzione ottima
- ¡E,F¿ è un matroide

19 Cos'è una struttura dati per insiemi disgiunti? Definire formalmente quali sono le operazioni principali su una struttura dati per insiemi disgiunti.