Algoritmi sui Grafi

Tutorato di Algoritmi e Laboratorio

Alessio Mezzina
PhD Student

DMI - UniCT

Giugno 2025

Contenuti

Outline

- Grafo $G = (V, E)$ orientato o non orientato
- Peso (o costo) di un arco: funzione $w : E \to \mathbb{R}$ (facoltativo)
- Grado di un vertice, cammino, ciclo, componente connessa (vediamo dopo)
- Obiettivo: rappresentare G in memoria in modo efficiente

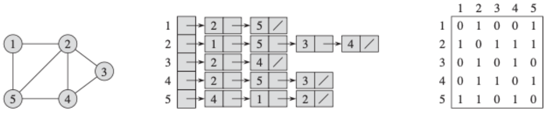
Liste di adiacenza

- Array `Adj` di $|V|$ liste, una per ciascun vertice
- In $\texttt{Adj}[u]$ compaiono tutti i vertici v tali che $(u, v) \in E$
- Memoria necessaria: $\Theta(|V| + |E|)$
- **Vantaggi**: rappresentazione compatta per grafi sparsi ($|E| \ll |V|^2$); iterazione sugli archi uscenti in tempo proporzionale al grado di u
- **Svantaggio**: per verificare se $(u, v) \in E$ occorre cercare v in $\texttt{Adj}[u]$

Matrice di adiacenza

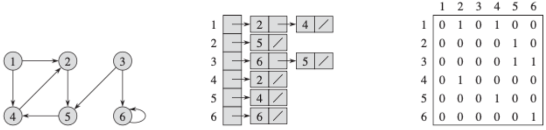
- Matrice $A \in \{0, 1\}^{|V| \times |V|}$ con $A_{ij} = 1$ se $(i, j) \in E$, 0 altrimenti
- Nei grafi non orientati la matrice è simmetrica $(A = A^{\mathsf{T}})$
- Memoria richiesta: $\Theta(|V|^2)$, indipendente da $|E|$
- **Vantaggi**: test $(u, v) \in E$ in $O(1)$; utile per grafi densi e negli algoritmi APSP basati su matrici (es. Floyd–Warshall)
- **Svantaggi**: spreco di spazio per grafi sparsi

Liste vs Matrici



(a) (b) (c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



(a) (b) (c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Grafi pesati e attributi

- Grafi pesati: sostituiamo il valore 1 con il peso $w(u, v)$ nella matrice, oppure memorizziamo la coppia $(v, w(u, v))$ in $\texttt{Adj}[u]$
- Attributi di vertici/archi (es. distanza, colore) possono essere tenuti in vettori paralleli ($\texttt{d}[u]$) o come campi in strutture/oggetti
- Memoria rimane $\Theta(|V| + |E|)$ per le liste e $\Theta(|V|^2)$ per la matrice

BFS: obiettivi e idea generale

- Dato un grafo $G = (V, E)$ e una sorgente s, esplora sistematicamente tutti i vertici raggiungibili da s.
- Calcola per ogni vertice v la distanza $d(v) =$ minimo numero di archi sul cammino da s a v (se irraggiungibile $d(v) = \infty$).
- Produce un *albero di visita in ampiezza* radicato in s: il cammino semplice da s a v nell'albero è un **shortest path** in G (rispetto al numero di archi).
- Funziona su grafi *orientati e non orientati*.
- Schema riutilizzato da algoritmi fondamentali (ad es. Prim per MST, Dijkstra per SSSP con pesi non negativi).

BFS: colori dei vertici e frontiera

- Ogni vertice è colorato **bianco**, **grigio** o **nero**:
  - bianco: non ancora scoperto;
  - grigio: scoperto ma la sua lista di adiacenza non è stata ancora scandita del tutto;
  - nero: completamente esplorato.
- I vertici grigi formano la **frontiera** tra la parte già visitata e quella ancora ignota.
- Se $(u, v) \in E$ e u è nero, allora v è grigio o nero $\Rightarrow$ *mai* esistono archi da un vertice nero verso uno bianco.

BFS: coda FIFO e attributi

- La frontiera è gestita con una coda FIFO Q: estrae sempre il vertice grigio più “antico”, garantendo l'ordine per distanza.
- Attributi mantenuti per ogni $u \in V$:
  - `u.color` – bianco, grigio, nero;
  - `u.d` – distanza da s (intero non negativo o ∞);
  - $\texttt{u.}\pi$ – predecessore nell'albero BFS (oppure NIL).
- Quando un vertice bianco v viene scoperto da u:

$$v.\texttt{color} \leftarrow \text{grigio}, \; v.\texttt{d} \leftarrow u.\texttt{d} + 1, \; v.\pi \leftarrow u, \; \text{enqueue}(Q, v)$$

BFS: correttezza e distanze minime

- Sia $\delta(s, v)$ la distanza minima (in numero di archi) da s a v in G.
- **Lemma** – Al termine di BFS:

$$\forall v \in V, \quad v.\mathtt{d} = \delta(s, v).$$

- **Idea della dimostrazione**: BFS scopre prima tutti i vertici a distanza 0, poi tutti quelli a distanza 1, poi 2, ecc. La coda FIFO assicura che un vertice venga estratto solo dopo che sono stati estratti *tutti* i vertici a distanza minore.
- L'albero BFS contiene per ogni v raggiungibile da s un cammino di lunghezza $\delta(s, v)$, quindi è uno *shortest-path tree* rispetto al conteggio degli archi.

BFS Pseudocode

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

BFS Example

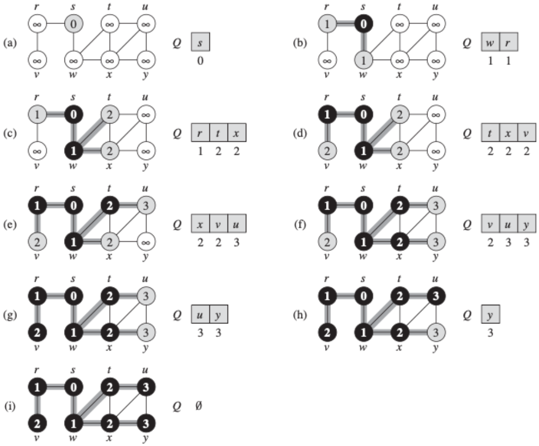


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex u. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the queue.

DFS: strategia di esplorazione

- Ricerca **in profondità**: finché possibile segue un arco inesplorato dall'ultimo vertice scoperto.
- Quando un vertice u non ha più adiacenti bianchi, il processo *fa backtracking* al suo predecessore.
- Se rimangono vertici bianchi, la visita riparte da uno di essi (nuova radice) finché l'intero grafo non è stato esplorato.
- Funziona su grafi orientati e non orientati.

Foresta DFS e colori dei vertici

- L'insieme dei predecessori forma una **foresta DFS**; gli archi predecessori sono *tree edges*.
- Stati dei vertici:

  bianco non ancora scoperto;
  grigio scoperto, ma con archi uscenti da scandire;
  nero esplorazione completa (lista di adiacenza esaminata).
- Proprietà: se $(u, v) \in E$ e u è nero, allora v è grigio o nero (mai bianco).

Timestamp di scoperta e termine

- Variabile globale `time` incrementata a ogni evento.
- Attributi:

$$v.\mathtt{d} = \text{tempo di scoperta}, \qquad v.\mathtt{f} = \text{tempo di termine}$$

  con $v.\mathtt{d} < v.\mathtt{f}$.
- Gli intervalli $[v.\mathtt{d}, v.\mathtt{f}]$ codificano la struttura gerarchica della visita (*figlio* $\subset$ intervallo *padre*).
- Utili per:
  - ordinamento topologico;
  - classificazione degli archi (tree, back, forward, cross);
  - componenti fortemente connesse (Kosaraju/Tarjan).

DFS: complessità e dipendenze dall'ordine

- Ogni vertice è colorato e terminato una sola volta; ogni arco esplorato al più una volta.
- **Tempo**: $\Theta(|V| + |E|)$ **Spazio**: $\Theta(|V|)$ per colori, timestamp, predecessori.
- L'ordine in cui si visitano:
  - i vertici in V (ciclo esterno);
  - gli adiacenti in `Adj`[u]

  può cambiare la foresta DFS ma non la correttezza né la complessità dell'algoritmo.

DFS PseudoCode

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)

DFS-VISIT(G, u)
time = time + 1                // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]          // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                // blacken u; it is finished
time = time + 1
u.f = time
```

DFS Example

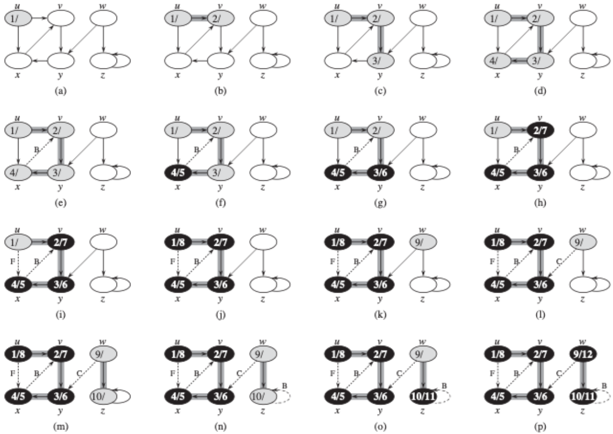


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

DFS: classificazione degli archi

- Durante la visita, ogni arco (u, v) ricade in **uno** dei seguenti tipi rispetto alla foresta DFS:

  Archi d'albero (tree): (u, v) scopre per la prima volta v.

  Archi all'indietro (back): collegano u a un *antenato* di u (self-loop incluso).

  Archi in avanti (forward): collegano u a un *discendente* di u distinto da un arco d'albero.

  Archi trasversali (cross): tutti gli altri; collegano nodi in sotto-alberi diversi o dello stesso livello senza rapporto antenato-discendente.

- Queste categorie aiutano a riconoscere proprietà del grafo (es. aciclicità nei digrafi).

Come determinare il tipo di un arco

- Quando esploriamo (u, v):

Colore di v	**Tipo di arco**
bianco	tree
grigio	back
nero	*forward oppure cross*

- Caso nero:

$$\begin{cases} u.d < v.d & \implies \text{forward} \\ u.d > v.d & \implies \text{cross} \end{cases}$$

  dove $x.d$ è il timestamp di scoperta.

- Nei **grafi non orientati** compaiono solo tree e back-edges (forward/cross non possono esistere).

Outline

Ordinamento topologico: definizione e contesto

- Un ordinamento topologico di un digrafo aciclico (DAG) $G = (V, E)$ è una disposizione lineare dei vertici tale che
$$(u, v) \in E \implies u \text{ precede } v.$$
- Utile per problemi di precedenza (compilazione, pianificazione, pipeline . . .).
- Esempio classico: ordinare i capi di vestiario del prof. Bumstead (calzini $\rightarrow$ scarpe, pantaloni prima della cintura, ecc.).
- Se G contiene un ciclo diretto $\Rightarrow$ nessun ordinamento lineare è possibile.

Topological-Sort via Depth–First Search

1. Esegui **DFS** su G e registra per ogni vertice v il tempo di fine $v.f$.
2. Al termine di ogni visita, inserisci v in testa a una lista concatenata.
3. Restituisci la lista: i vertici appaiono in **ordine decrescente** di $v.f$.

Correttezza

Se $(u, v) \in E$, DFS garantisce $v.f < u.f \Rightarrow u$ compare prima di v nella lista.

Complessità

$$\Theta(|V| + |E|) \quad \text{(tempo e spazio, dominati dalla DFS).}$$

Outline

Cammini minimi: motivazione e modello

- Dato un digrafo *pesato* $G = (V, E)$ con funzione dei pesi $w : E \to \mathbb{R}$.
- Esempio: mappa stradale (nodi = incroci, archi = tratti di strada, peso = distanza, tempo o costo).
- Obiettivo generico:

  Trovare il cammino a peso minimo fra due (o più) vertici.

- Applicazioni: routing, pianificazione, reti di progetto, bioinformatica, robotica . . .

Definizioni formali

- **Peso di un cammino**
  Per $p = \langle v_0, v_1, \ldots, v_k \rangle$:

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

- **Peso minimo** (*shortest-path weight*)

$$\delta(u, v) = \begin{cases} \min\limits_{p: u \rightsquigarrow v} w(p) & \text{se esiste un cammino,} \\ \infty & \text{altrimenti.} \end{cases}$$

- **Cammino minimo** da u a v: qualsiasi cammino p con $w(p) = \delta(u, v)$.

Varianti del problema

- Single-source (SSSP): da s a tutti i vertici. $\rightarrow$ focus di questo capitolo.
- Single-destination: verso t da tutti i vertici (basta invertire gli archi).
- Single-pair: da u a v (stesso costo asintotico di SSSP).
- All-pairs (APSP): da ogni u a ogni v (algoritmi dedicati, vedi Cap. 25).

Pesi negativi e cicli a peso negativo

- I pesi possono essere negativi (penalità, credito, guadagno-costo).
- Se esiste un **ciclo a peso negativo** raggiungibile da s, le distanze non sono ben definite:

$$\delta(s, v) = -\infty \quad \text{per ogni } v \text{ raggiungibile dal ciclo.}$$

- Algoritmi:

  Dijkstra richiede $w(e) \geq 0$.

  Bellman–Ford ammette pesi negativi e *rileva* cicli a peso negativo.

Cicli nei cammini minimi

- Un cammino minimo non contiene cicli a peso positivo (li si può eliminare e ottenere un cammino più breve).
- Nemmeno cicli negativi, per definizione.
- I cicli a peso 0 possono essere rimossi $\Rightarrow$ esiste sempre un cammino minimo *semplice* di al più $|V| - 1$ archi.

Predecessori e albero dei cammini minimi

- Ogni vertice v mantiene $\texttt{v.}\pi$ (predecessore) e `v.d` (stima di distanza).
- Il sottografo indotto dalle predecessori

$$G_\pi = (V_\pi, E_\pi), \quad V_\pi = \{v : \pi(v) \neq \mathrm{NIL}\} \cup \{s\}$$

costituisce un **shortest-paths tree** quando per ogni v raggiungibile da s la strada in G_π è un cammino minimo in G.

Inizializzazione e rilassamento

Algorithm 1 INITIALIZE-SINGLE-SOURCE(G, s)

```
for ogni v ∈ V do
    v.d ← +∞,    v.π ← NIL
end for
s.d ← 0
```

Algorithm 2 RELAX(u, v, w)

```
if v.d > u.d + w(u, v) then
    v.d ← u.d + w(u, v)
    v.π ← u
end if
```

- **Rilassare** un arco può solo diminuire le stime $v.d$.
- Gli algoritmi SSSP differiscono nel numero e nell'ordine delle rilassamenti.

Bellman–Ford: panoramica

- Risolve SSSP su grafi con pesi negativi
- Output:
  1. **Boolean** – `TRUE` se non c'è alcun ciclo negativo raggiungibile da s, `FALSE` altrimenti.
  2. Distanze $\delta(s, v)$ e predecessori che formano un *shortest-paths tree* se l'esito è `TRUE`.
- Idea chiave: rilassare tutti gli archi $|V| - 1$ volte per far "diffondere" le distanze corrette lungo cammini di lunghezza al più $|V| - 1$ archi.

Passi principali dell'algoritmo

1. **Initialize-Single-Source**(G, s):
   $s.d \leftarrow 0,\;\; v.d \leftarrow +\infty$ e $v.\pi \leftarrow$ NIL per $v \neq s$.
2. **Fase di rilassamento** $(|V| - 1$ iterazioni):
   for $(u, v) \in E$ **do** RELAX(u, v, w).
3. **Verifica cicli negativi**:
   se esiste (u, v) con $v.d > u.d + w(u, v) \;\Rightarrow$ ciclo a peso < 0 raggiungibile.

Osservazione: dopo k-esima iterazione, tutte le distanze corrette su cammini di $\leq k$ archi sono state propagate.

Correttezza: idea della dimostrazione

- **Invariante**: dopo i iterazioni di rilassamento, $v.d \leq \delta(s, v)$ per ogni v, e per cammini di $\leq i$ archi vale l'uguaglianza.
- Dopo $|V| - 1$ iterazioni ogni cammino semplice (al più $|V| - 1$ archi) è stato considerato $\Rightarrow \; v.d = \delta(s, v)$.
- Test finale: se una distanza può ancora diminuire, esiste un ciclo negativo raggiungibile da s. (Una catena di $|V|$ rilassamenti implicherebbe percorso $\geq |V|$ archi $\Rightarrow$ qualche ciclo.)

Complessità ed applicazioni

- **Tempo**: $O(|V| \cdot |E|)$ (init $+$ $(|V| - 1)$ pass $+$ test finale).
- **Spazio**: $O(|V|)$ per distanze, predecessori e coda di archi.
- Utilizzabile quando:
  - pesi negativi ma assenza (o rilevazione) di cicli negativi;
  - si vuole calcolare potenziali per trasformare i pesi (alg. di Johnson).
- Su grafi sparsi piccoli o medi può competere con Dijkstra (implementazione semplice).

Algorithm 3 Bellman–Ford(G, w, s)

```
Initialize-Single-Source(G, s)
for i = 1 to |V| − 1 do
    for ogni (u, v) ∈ E do
        Relax(u, v, w)
    end for
end for
for ogni (u, v) ∈ E do
    if d[v] > d[u] + w(u, v) then
        return FALSE                ▷ ciclo negativo trovato
    end if
end for
return TRUE
```

Example

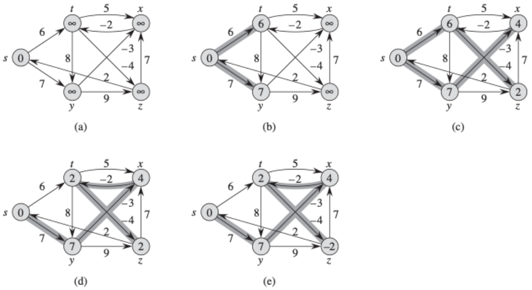


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

DAG Shortest Paths

- Un grafo aciclico orientato permette topological sort in $O(V + E)$
- Una volta ordinati, basta rilassare gli archi in ordine topologico
- Complessità totale: $O(V + E)$

DAG Shortest Paths: pseudocodice

Algorithm 4 DAG-Shortest-Paths(G, w, s)

```
Topological-Sort(G)
Initialize-Single-Source(G, s)
for ogni u nell'ordine topologico do
    for ogni (u, v) ∈ E do
        Relax(u, v, w)
    end for
end for
```

Dijkstra: requisiti e idea

- Input: grafo orientato $G = (V, E)$ con **pesi non negativi**, sorgente s.
- Mantiene un insieme S di vertici con distanza definitiva $\delta(s, v)$ già stabilita.
- Passo chiave (greedy): estrai da una *min-priority queue* il vertice $u \notin S$ con stima minima $u.d$; quel valore è già ottimo $\Rightarrow$ inserisci u in S e rilassa gli archi uscenti.

Algoritmo (scheletro)

1. INITIALIZE-SINGLE-SOURCE(G, s).
2. $S \leftarrow \varnothing, \quad Q \leftarrow V$ (coda di priorità su $v.d$).
3. **while** $Q \neq \varnothing$
    1. $u \leftarrow$ EXTRACT-MIN(Q)
    2. $S \leftarrow S \cup \{u\}$
    3. **for** $(u, v) \in E$ **do** RELAX(u, v, w)

Invariante: prima di ogni estrazione S contiene i vertici con distanza definitiva, $Q = V \setminus S$.

Correttezza (intuizione)

- Pesi $\geq 0 \Rightarrow$ i cammini che passano per vertici già in S non possono migliorare le stime di vertici ancora in Q.
- Quando u è estratto, qualsiasi cammino $s \rightsquigarrow v$ con $v \in Q$ ha peso $\geq u.d$; quindi $u.d = \delta(s, u)$.
- Dopo $|V|$ estrazioni tutte le distanze sono definitive e i predecessori formano lo *shortest-paths tree*.

Implementazioni e complessità

Struttura della coda	Tempo	Spazio
Array lineare	$O(\|V\|^2)$	$O(\|V\|)$
Heap binario	$O((\|V\| + \|E\|) \log \|V\|)$	$O(\|V\|)$
Fibonacci heap	$O(\|V\| \log \|V\| + \|E\|)$	$O(\|V\|)$

Applicazioni e limitazioni

- Navigazione stradale, routing IP (OSPF), robot path planning.
- Pre-processing in algoritmo di Johnson per APSP.
- Limite: fallisce con pesi negativi (può restituire distanze errate).
- Varianti:
  - *A**: aggiunge euristica ammissibile per velocizzare la ricerca.
  - *Dial's* e *Radix heap*: ottimizzazioni quando i pesi sono interi piccoli.

Outline

APSP via programmazione dinamica (min-plus)

- Vogliamo le distanze minime tra *tutte* le coppie di vertici in un digrafo pesato $G = (V, E)$, $n = |V|$.
- Idea: calcolare iterativamente $L^{(m)} = (\ell_{ij}^{(m)})$ dove

$$\ell_{ij}^{(m)} = \text{peso minimo di un cammino } i \to j \text{ con } \leq m \text{ archi.}$$

- Osservazione chiave (Lemma 24.1): *ogni sottocammino di un cammino minimo è minimo* $\Rightarrow$ cammino minimo ha $\leq n - 1$ archi.

Ricorrenza ($min, +$ "moltiplicazione")

$$\ell_{ij}^{(0)} = \begin{cases} 0 & i = j, \\ +\infty & i \neq j, \end{cases} \qquad \ell_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ \ell_{ik}^{(m-1)} + w_{kj} \right\}.$$

- L'operazione è analoga alla moltiplicazione di matrici, ma con $+ \to \min$ e $\times \to +$ (**semianello min-plus**).
- Definiamo il prodotto min-plus $A \otimes B$: $(A \otimes B)_{ij} = \min_k \{a_{ik} + b_{kj}\}$.

Algorithm 5

```
for i = 1 to n do
    for j = 1 to n do
        ℓ'_ij ← +∞
        for k = 1 to n do
            ℓ'_ij ← min{ℓ'_ij, ℓ_ik + w_kj}
        end for
    end for
end for
```

Tempo $\Theta(n^3)$

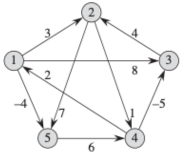
Algoritmo Slow-All-Pairs-Shortest-Paths

1. $L^{(1)} \leftarrow W$.
2. **for** $m = 2$ **to** $n - 1$ **do**
   $L^{(m)} \leftarrow L^{(m-1)} \otimes W$.

$$L^{(n-1)} = W^{\otimes(n-1)} \implies \delta(i,j) = \ell_{ij}^{(n-1)}.$$

$$\boxed{\Theta(n^4) \text{ tempo}, \ \ \Theta(n^2) \text{ spazio}}$$

Example



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figure 25.1 A directed graph and the sequence of matrices $L^{(m)}$ computed by SLOW-ALL-PAIRS-SHORTEST-PATHS. You might want to verify that $L^{(5)}$, defined as $L^{(4)} \cdot W$, equals $L^{(4)}$, and thus $L^{(m)} = L^{(4)}$ for all $m \geq 4$.

Accelerazione con *repeated squaring*

- Il prodotto min-plus è *associativo* $\Longrightarrow$ possiamo usare le potenze:

$$W^{\otimes 2}, W^{\otimes 4}, W^{\otimes 8}, \ldots$$

- Basta calcolare $W^{\otimes 2^{\lceil \log_2 (n-1) \rceil}}$ con **dlg (**$n - 1$**) e** moltiplicazioni.

Faster-All-Pairs-Shortest-Paths

Algorithm 6

1: $L \leftarrow W, \quad m \leftarrow 1$
2: **while** $m < n - 1$ **do**
3: $\quad L \leftarrow L \otimes L$ $\qquad \triangleright\ L \leftarrow W^{\otimes 2m}$
4: $\quad m \leftarrow 2m$
5: **end while**
6: **return** L

$$\boxed{\Theta(n^3 \log n) \text{ tempo}, \quad \Theta(n^2) \text{ spazio}}$$

Floyd–Warshall: idea di base

- Vertici numerati $1, \dots, n$.
- $d_{ij}^{(k)}$ = peso minimo di un cammino $i \to j$ i cui *vertici intermedi* appartengono all'insieme $\{1, \dots, k\}$ (estremi esclusi).
- Caso $k = 0$: nessun intermedio $\Rightarrow$ $d_{ij}^{(0)} = w_{ij}$.
- Caso generale ($k \geq 1$):

$$d_{ij}^{(k)} = \min\left\{\, d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\}.$$

  - Se il cammino minimo *non* usa k, resta $d_{ij}^{(k-1)}$.
  - Se lo usa, si concatena il cammino $i \to k$ con $k \to j$ (entrambi con intermedi $\leq k - 1$).

Algoritmo Floyd–Warshall

Algorithm 7 FLOYD-WARSHALL(W)

```
D ← W                                   ▷ d_ij^(0) = w_ij
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            D_ij ← min{D_ij, D_ik + D_kj}
        end for
    end for
end for
return D                                ▷ d_ij^(n) = δ(i, j)
```

$$\Theta(n^3) \text{ tempo}, \qquad \Theta(n^2) \text{ spazio.}$$

Interpretazione matriciale

- Ogni iterazione k “sblocca” il vertice k come possibile intermedio.
- La tripla `for` corrisponde a un *min-plus update* sull’intera matrice.
- Dopo il passo k, la riga i e la colonna j tengono conto dei cammini che possono passare per k.

Costruzione dei cammini minimi

- Manteniamo anche la matrice dei predecessori $\Pi^{(k)}$:

$$\pi_{ij}^{(0)} = \begin{cases} \text{nil} & i = j \text{ o } w_{ij} = +\infty, \\ i & \text{altrimenti.} \end{cases}$$

- Aggiornamento:

$$\text{se } D_{ik} + D_{kj} < D_{ij} \text{ allora } \begin{cases} D_{ij} \leftarrow D_{ik} + D_{kj} \\ \pi_{ij} \leftarrow \pi_{kj} \end{cases}$$

- A fine algoritmo, Π permette Print-Path(i, j) in $O(\ell)$ (dove ℓ = numero archi del cammino).

Riepilogo e confronti

- **Pro**:
  - algoritmo compatto, costante nascosta piccola;
  - gestisce pesi negativi (ma *non* cicli negativi).
- **Contro**:
  - complessità $O(n^3) \Rightarrow$ adatto a grafi densi o $n \lesssim 5000$.
  - spazio $O(n^2)$.
- Alternativa per grafi sparsi: algoritmo di Johnson $O(n^2 \log n + n|E|)$.

Riferimenti bibliografici

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd Ed., MIT Press, 2009.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, 3/ed, McGraw-Hill Italia, 2010.