Grafi

Salvatore Filippone
salvatore.filippone@uniroma2.it

Un grafo $\mathcal{G}$ è una collezione di due insiemi, i *vertici* o *nodi* $\mathcal{V}$ e gli *archi* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ (ossia, coppie di nodi (I, J)):

Grafi orientati: sono grafi in cui gli archi (I, J) e (J, I) sono distinti (in particolare, uno dei due potrebbe non esistere);

Grafi non orientati: grafi in cui non si considera l'ordine dei nodi in un arco, ossia $[I, J]$ corrisponde sia a (I, J) che a (J, I).

Nodi adiacenti: due nodi i e j si dicono *adiacenti* se esiste un arco che li connette $(i, j) \in \mathcal{E}$;

Arco incidente: l'arco (i, j) si dice *incidente* su j (da i);

Grado di un nodo: il grado in entrata è il numero di archi incidenti su di un nodo (in uscita: archi incidenti da un nodo)

Cammino: Dato grafo orientato $\mathcal{G}$, un *cammino* di lunghezza k è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in \mathcal{E}$. Se non ci sono nodi ripetuti, il cammino si dice *semplice*, se $u_o = u_k$ si dice *chiuso*; se è sia semplice che chiuso allora è un *ciclo*;

Grafi orientati aciclici: Grafi in cui *non esiste alcun ciclo*

$$(l_1, l_2), (l_2, l_3), \ldots (l_n, l_1).$$

Catena: In un grafo NON orientato $\mathcal{G}$, una *catena* è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tale che $[u_i, u_{i+1}] \in \mathcal{E}$; analogamente con la definizione per grafi orientati, una catena semplice e chiusa è un *circuito*.

Connessione forte

Se $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ è un grafo orientato, allora $\mathcal{G}$ è *fortemente connesso* se per *ogni* coppia di nodi u, v esiste un cammino da u a v, ed un cammino da v a u.

Componenti fortemente connesse

Si dice *componente fortemente connessa* un sottografo fortemente connesso e massimale:

- Un grafo $\mathcal{G}'$ è un *sottografo* di $\mathcal{G}$ se e solo se $\mathcal{V}' \subseteq \mathcal{V}$ e $\mathcal{E}' \subseteq \mathcal{E}$;
- Un sottografo $\mathcal{G}'$ è *massimale* se non esiste nessun sottografo fortemente connesso di $\mathcal{G}$ che lo contiene strettamente, ossia non esiste $\mathcal{G}''$ tale che $\mathcal{G}' \subset \mathcal{G}'' \subseteq \mathcal{G}$;

Connessione (semplice)

Se $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ è un grafo NON orientato, allora $\mathcal{G}$ è *connesso* se per ogni coppia di nodi u, v esiste una catena da u a v.

Componenti connesse

Si dice *componente connessa* un sottografo connesso e massimale:

Un grafo non orientato è un *albero libero* se è connesso e per ogni coppia di nodi esiste una e una sola catena semplice.
Un albero *radicato* si ottiene da un albero libero fissando arbitrariamente un nodo come radice, e poi ordinando i rimanenti per livelli.

Operatori su Grafi

Graph() Crea un grafo vuoto;

insertNode(u) Inserisci il nodo u;

insertEdge(u,v) Inserisci l'arco (u, v);

deleteNode(u) Cancella il nodo u;

deleteEdge(u,v) Cancella l'arco (u, v);

adj(u) Restituisce l'insieme dei nodi adiacenti a u;

V() Restituisce l'insieme di tutti i nodi.

Denotiamo $m = |\mathcal{E}|$, $n = |\mathcal{V}|$

Implementazione con matrice di adiacenza

Si rappresenta il grafo con una matrice M di dimensione $n \times n$ tale che

$$M_{ij} = \begin{cases} 1 & \text{se } (i,j) \in \mathcal{E} \\ 0 & \text{se } (i,j) \notin \mathcal{E} \end{cases}$$

- Verificare l'appartenenza di un arco al grafo ha costo $O(1)$;
- `adj(u)` ha costo $\Theta(n)$.
- Se gli archi sono pesati, allora si memorizza il peso (oppure ∞) nell'elemento corrispondente

Denotiamo $m = |\mathcal{E}|$, $n = |\mathcal{V}|$

Implementazione con matrice di incidenza nodi-archi

M di dimensione $n \times m$ tale che

$$M_{ij} = \begin{cases} -1 & \text{se arco } (j) \text{ esce dal nodo } i \\ 1 & \text{se arco } (j) \text{ entra nel nodo } i \\ 0 & \text{altrimenti} \end{cases}$$

Utile in problemi di programmazione lineare, ma costosa in memoria e in tempo.

Denotiamo $m = |\mathcal{E}|$, $n = |\mathcal{V}|$

Implementazione con liste di adiacenza

Un array di dimensione n dove per ogni nodo u viene predisposta la lista $adj(u)$ (di solito realizzata con puntatori).

- Occupazione di memoria ottimale $\Theta(n + m)$;
- Scansione della lista di adiacenza ottimale;
- Scansione degli archi ottimale;
- Verifica della appartenenza di un arco $O(|adj(u)|)$ (non ottimale).

Visita di un grafo

Definiamo una generica "visita" di un grafo, in cui vengono esaminati tutti i nodi e tutti gli archi:

Algorithm 1: Visita(Grafo G, nodo r)

```
SET S ← Set();
S.insert(r);
marca nodo r come "trovato";
while S.size() > 0 do
    NODE u ← S.remove();
    esamina il nodo u;
    foreach v ∈ G.adj(u) do
        Esamina l'arco (u, v);
        if v non è già "trovato" then
            marca nodo v come "trovato";
            S.insert(v);
```

Visita BFS di un grafo

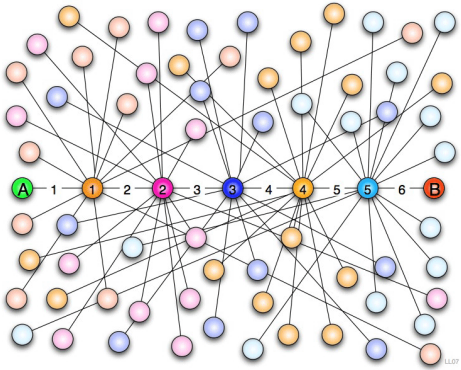
Visita per "livelli":

Algorithm 2: BFS(Grafo G, nodo r)

```
QUEUE S ← Queue();
S.enqueue(r);
boolean[] visitato ← new boolean[1 ... G.n];
visitato[r] ← true;
while not S.isEmpty() do
    NODE u ← S.dequeue();
    esamina il nodo u;
    foreach v ∈ G.adj(u) do
        Esamina l'arco (u, v);
        if not visitato[v] then
            visitato[v] ← true;
            S.enqueue(v);
```

Sei gradi di separazione

La teoria dei sei gradi di separazione in semiotica e in sociologia è un'ipotesi secondo la quale ogni persona può essere collegata a qualunque altra persona o cosa attraverso una catena di conoscenze e relazioni con non più di 5 intermediari.



Esempio: il numero di Erdős

Data la relazione "A ha scritto un articolo con B", quale è la distanza (numero di livelli) da Erdős Pál, detta anche *numero di Erdős*?

Esempio: il numero di Erdős

Data la relazione "A ha scritto un articolo con B", quale è la distanza (numero di livelli) da Erdős Pál, detta anche *numero di Erdős*? Statistica dei numeri di Erdős (*Physics Central*, 2009):

0 1 persona
1 504 persone
2 6593 persone
3 33605 persone
4 83642 persone
5 87760 persone
6 40014 persone
7 11591 persone
8 3146 persone
9 819 persone
10 244 persone
11 68 persone
12 23 persone
13 5 persone

Esempio: il numero di Erdős

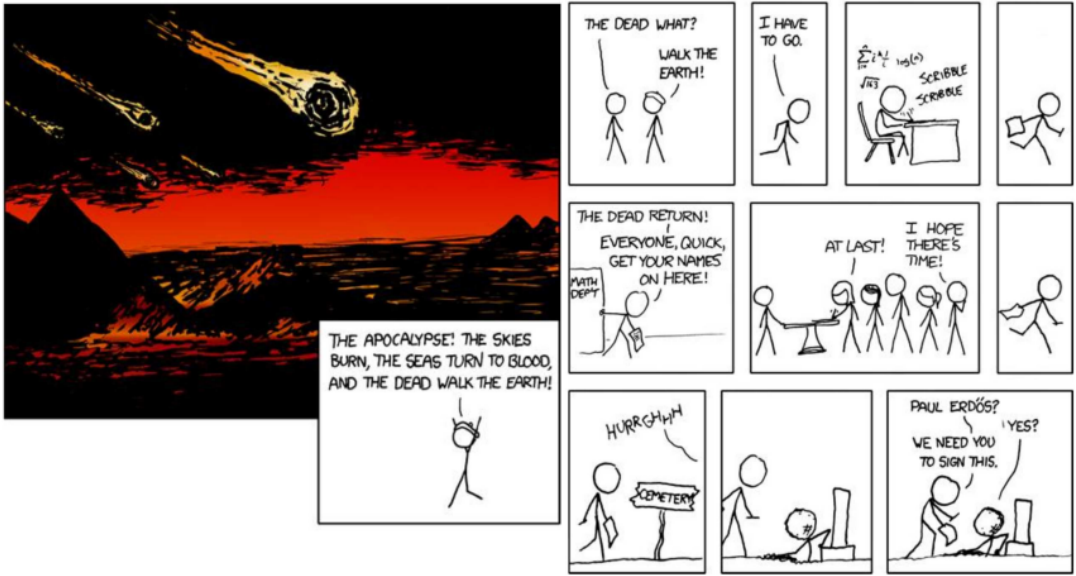
Data la relazione "A ha scritto un articolo con B", quale è la distanza (numero di livelli) da Erdős Pál, detta anche *numero di Erdős*? Statistica dei numeri di Erdős (*Physics Central*, 2009):

0 1 persona
1 504 persone
2 6593 persone
3 33605 persone
4 83642 persone
5 87760 persone
6 40014 persone
7 11591 persone
8 3146 persone
9 819 persone
10 244 persone
11 68 persone
12 23 persone
13 5 persone

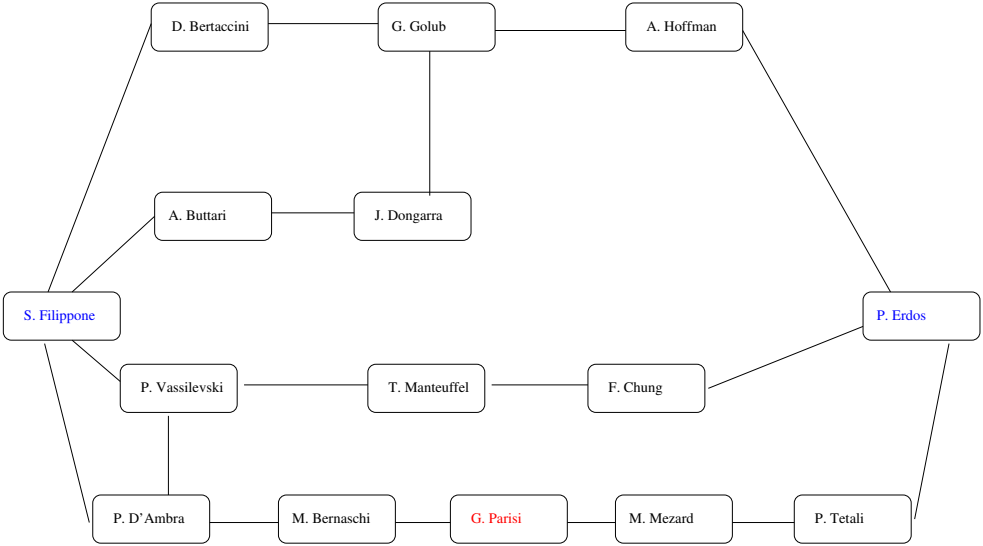
Numeri di Erdős di persone famose:

Albert Einstein: 2;
Donald Knuth: 2;
Linus Torvalds: ∞;
Jack Dongarra: 3;
Gene Golub: 2;
Brian Kernighan: 3;
William Gates: 4;
Piermarco Cannarsa: 2;
Giorgio Parisi: 3;
Enrico Fermi: 3;

Esempio: il numero di Erdős



Esempio: il numero di Erdős



Esempio: il numero di Erdős

Algorithm 3: erdos(GRAFO G, NODO r, **integer**[] erdős, NODO[] p)

```
QUEUE S ← Queue();
S.enqueue(r);
foreach v ∈ G.V() do
    erdős[v] ← ∞;
erdős[r] ← 0;
p[r] ← nil;
while not S.isEmpty() do
    NODE u ← S.dequeue();
    foreach v ∈ G.adj(u) do
        if erdős[v] = ∞ then
            erdős[v] ← erdős[u] + 1;
            p[v] ← u;
            S.enqueue(v);
```

La procedura restituisce anche un vettore contenente una lista dei "padri" che consente di ricostruire il percorso da Erdős al nodo desiderato

Una visita in profondità in preordine può essere impiegata per il calcolo delle componenti connesse di un grafo

Algorithm 4: integer [] cc(GRAFO G, STACK S)

```
integer[] id ← new integer[1...G.n];
foreach u ∈ G.V do
    id[u] ← 0;
integer conta ← 0;
while not S.isEmpty() do
    u ← S.pop();
    if id[u] = 0 then
        conta ← conta + 1;
        ccdfs(G, conta, u, id);
Output: id
```

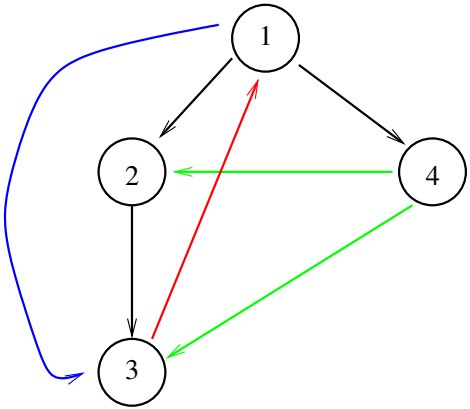
Algorithm 5: ccdfs(GRAFO G, **integer** $conta$, NODO u, **integer**[] id)

```
id[u] ← conta;
foreach v ∈ G.adj(u) do
    if id[v] = 0 then
        ccdfs(G, conta, v, id);
Output: id
```

La procedura si può modificare per generare l'albero ricoprente: ogni volta che $id[v] = 0$ in ccdfs si aggiunge l'arco (u, v) all'albero di indice $conta$ (se il grafo non è connesso si ottengono più alberi, ossia una *foresta*)

Albero ricoprente

- Se un arco non è incluso nell'albero ricoprente T, allora:
  - Se l'arco passa da un nodo di T ad un suo antenato, è un arco *all'indietro*;
  - Se l'arco passa da un nodo di T ad un suo discendente, allora è *in avanti*;
  - Altrimenti è *di attraversamento*
- Se il grafo non è orientato, allora l'arco appartiene a T oppure è all'indietro.



Si può costruire una procedura DFS generica che serve come schema per diversi scopi; si usano i vettori *dt* e *ft* per registrare quando vengono visitati i vari nodi.

DFS-schema

Algorithm 6: dfs-schema(GRAFO G, NODO u)

pre-visita nodo u;
$time \leftarrow time + 1$; $dt[u] \leftarrow time$;
foreach $v \in G.adj(u)$ **do**
    esamina arco (u, v);
    if $dt[v] = 0$ **then**
        esamina arco (u, v) in T;
        dfs-schema(G, v);
    else if $dt[u] > dt[v]$ **and** $ft[v] = 0$ **then**
        esamina arco (u, v) all'indietro;
    else if $dt[u] < dt[v]$ **and** $ft[v] \neq 0$ **then**
        esamina arco (u, v) in avanti;
    else
        esamina arco (u, v) di attraversamento;
post-visita nodo u;
$time \leftarrow time + 1$; $ft[u] \leftarrow time$;

Esempio: verifica se grafo è aciclico

Algorithm 7: ciclico(GRAFO G, NODO u)

$time \leftarrow time + 1$; $dt[u] \leftarrow time$;
foreach $v \in G.adj(u)$ **do**
  if $dt[v] = 0$ **then**
    if $ciclico(G, v)$ **then**
      Output: true
  else if $dt[u] > dt[v]$ **and** $ft[v] = 0$ **then**
    Output: true
$time \leftarrow time + 1$; $ft[u] \leftarrow time$;
Output: false

Un grafo orientato aciclico ammette un *ordinamento topologico*, ossia una sequenza $v_1, v_2, \ldots, v_n$ dove $i < j$ implica che v_i precede v_k.

Esempio

È dato un insieme di attività che devono rispettare dei vincoli; per la ricetta di una torta ad esempio:

1. Preparazione dell'impasto;
2. Riposo/lievitazione dell'impasto;
3. Preparazione della farcia;
4. Riposo della farcia;
5. Preparazione della glassa;
6. Cottura dell'impasto;
7. Raffreddamento della base;
8. Farcitura;
9. Glassatura.

Le relazioni di precedenza sono $(1,2)$, $(3,4)$, $(2,6)$, $(6,7)$, $(7,8)$, $(4,8)$, $(8,9)$, $(5,9)$; un ordinamento topologico possibile è quello visto, un altro è $(3,4,1,2,6,7,8,5,9)$.

Esempio: `make`

Topological sorting

Algorithm 8: topsort(GRAFO G, STACK S)

```
boolean[] visitato ← boolean[1..G, n];
foreach u ∈ G.V() do
    visitato[u] ← false;
foreach u ∈ G.V() do
    if not visitato[u] then
        ts-dfs(G, u, visitato, S);
```

Algorithm 9: ts-dfs(GRAFO G, NODO u, **boolean**[] *visitato*, STACK S)

```
visitato[u] ← true;
foreach v ∈ G.adj(u) do
    if not visitato[v] then
        ts-dfs(G, v, visitato, S);
S.push(u);
```

Definizione

Dato un grafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ con pesi $w(u, v)$ non negativi, trovare un albero $\mathcal{G}' = (\mathcal{V}, \mathcal{T})$ (quindi con $n - 1$ archi) tale che la funzione

$$\sum_{(u,v)\in\mathcal{T}} w(u, v)$$

sia minimizzata.

Gli algoritmi di Kruskal e Prim consentono di risolvere questo problema in un tempo[1] $O(m \log(n))$, e sono esempi di algoritmi "greedy" (golosi).

[1] $m = |\mathcal{E}|$, $n = |\mathcal{V}|$

Algoritmo di Prim

Algorithm 10: prim(GRAFO G, NODO r, **integer** [] p)

PRIORITYQUEUE $Q \leftarrow MinPriorityQueue()$;
PRIORITYITEM[] $pos \leftarrow$ **new**PRIORITYITEM$[1..G-n]$;
$visitato[u] \leftarrow$ **true**;
foreach $u \in G.V() - \{r\}$ **do**
  $pos[u] \leftarrow Q.insert(u, +\infty)$;
$pos[r] \leftarrow Q.insert(r, 0)$;
$p[r] \leftarrow 0$;
while not $Q.isEmpty()$ **do**
  NODO $u \leftarrow Q.DeleteMin()$;
  $pos[u] \leftarrow$ **nil**;
  foreach $v \in G.adj(u)$ **do**
    if $pos[v] \neq$ **nil and** $w(u,v) < pos[v].priority$ **then**
      $Q.decrease(pos[v], w(u,v))$;
      $p[v] \leftarrow u$;

Cammini minimi

Dato $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, un *costo* associato ad ogni arco $w(u, v)$. ed un cammino $c = v_1, \ldots, v_k$ definiamo il costo del cammino:

$$w(c) = \sum_{i=2}^{k} w(v_{i-1}, v_i).$$

Problema dei cammini minimi

Dato un grafo orientato $\mathcal{G}$ ed un nodo r, trovare per ogni nodo u un cammino di costo minimo

Una soluzione *ammissibile* consiste nel trovare un albero di ricopertura T.

Teorema (di Bellman)

Una soluzione ammissibile T è ottima se e solo se:

- *Per ogni arco $(u, v) \in T$ vale $d_v = d_u + w(u, v)$;*
- *Per ogni arco $(u, v) \in E$, vale $d_v \leq d_u + w(u, v)$;*

Algoritmo prototipo

Algorithm 11: prototipoCamminiMinimi(GRAFO G, NODO r)

% Inizializza T ad una foresta di copertura di nodi isolati ;
% Inizializza d con una sovrastima della distanza (0 per r, ∞ altruimenti);
while $\exists(u, v) : d_u + (u, v) < d$ **do**
    $d \leftarrow d_u + (u, v)$;
    % Sostituisci il padre di v in T con u;

Algoritmo prototipo (parte 1)

Algorithm 12: camminiMinimi(GRAFO G, NODO r, **integer**[] T)

integer[] $d \leftarrow$ **new integer**[1...$G.n$];
boolean[] $b \leftarrow$ **new boolean**[1...$G.n$];
foreach $u \in G.V - \{r\}$ **do**
    $T[u] \leftarrow$ **nil**;
    $d[u] \leftarrow \infty$;
    $b[u] \leftarrow$ **false**;
$T[r] \leftarrow$ **nil**;
$d[r] \leftarrow 0$;
$b[r] \leftarrow$ **true**;
SET $S \leftarrow Set()$;
$S.aggiungi(r)$;

Algoritmo prototipo (parte 2)

```
while not S.isEmpty() do
    NODO u ← S.estrai();
    b[u] ← false;
    foreach v ∈ G.adj(u) do
        if d[u] + w(u, v) < d[v] then
            if not b[v] then
                S.aggiungi(v);
                b[v] ← true;
            else
                % Azione se v è già in S;
            T[v] ← u;
            d[v] ← d[u] + w(u, v);
```

Nell'algoritmo di Dijkstra per S si usa una coda con priorità:

Algoritmo di Dijkstra (parte 1)

Algorithm 13: Dijkstra(GRAFO G, NODO r, **integer**[] T)

integer[] $d \leftarrow$ **new integer**[1...$G.n$];
boolean[] $b \leftarrow$ **new boolean**[1...$G.n$];
foreach $u \in G.V - \{r\}$ **do**
    $T[u] \leftarrow$ **nil**;
    $d[u] \leftarrow \infty$;
    $b[u] \leftarrow$ **false**;
$T[r] \leftarrow$ **nil**;
$d[r] \leftarrow 0$;
$b[r] \leftarrow$ **true**;
PRIORITYQUEUE $S \leftarrow$ *PriorityQueue*();
$S.insert(r, 0)$;

Algoritmo di Dijkstra (parte 2)

```
while not S.isEmpty() do
    NODO u ← S.deleteMin();
    b[u] ← false;
    foreach v ∈ G.adj(u) do
        if d[u] + w(u, v) < d[v] then
            if not b[v] then
                S.insert(v, d[u] + w(u, v));
                b[v] ← true;
            else
                S.decrease(v, d[u] + w(u, v));
            T[v] ← u;
            d[v] ← d[u] + w(u, v);
```

Nell'algoritmo di Bellman-Ford-Moore per S si usa una coda (semplice):

Algoritmo di Bellman-Ford-Moore

N.b.: funziona anche con pesi negativi!

Algorithm 14: BellmanFordMoore(GRAFO G, NODO r, **integer**[] T)

```
...;
QUEUE S ← Queue();
S.enqueue(r, 0);
while not S.isEmpty() do
    NODO u ← S.dequeue();
    b[u] ← false;
    foreach v ∈ G.adj(u) do
        if d[u] + w(u, v) < d[v] then
            if not b[v] then
                S.enqueue(v, d[u] + w(u, v));
                b[v] ← true;
            T[v] ← u;
            d[v] ← d[u] + w(u, v);
```

Dato un grafo

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

trovare la lunghezza del cammino minimo tra ogni possibile coppia di nodi. Formulazione di "forza bruta" — Floyd-Warshall: *simile ad un prodotto di matrici generalizzato su* $(R, \min, +)$ *invece che* $(R, +, \times)$

Algorithm 15: Floyd-Warshall APSP

```
Ms ← Ma;
for p = log(min(|V|, |E|)) do
    for D(i, j) ∈ Ms (in parallel) do
        for k = 1, ... do
            D(i, j) ← min (D(i, j), D(i, k) + D(k, j));
```