4 Algoritmi su Grafi

Un grafo orientato/diretto G e' una coppia (V, E) dove $E \subseteq V \times V$. Gli elementi di V si chiamano nodi/vertici mentre gli elementi di E si chiamano archi. Si riporta la terminologia e notazione che useremo in relazione ai grafi (vedi anche Sezione 5.4 del Cormen):

- $e: u \to v$ significa che e e' un arco da u a v, cioe' $e = (u, v) \in E$. Si dice anche che e esce da u ed entra in v. Un arco $e: u \to u$ e' detto **cappio**.
- v **adiacente** ad u significa che $(u, v) \in E$.
- ad un nodo u sono associati i seguenti numeri naturali:

  grado entrante (in-degree) = numero di archi entranti in u
  grado uscente (out-degree) = numero di archi uscenti da u
  grado = grado entrante + grado uscente.
- un **cammino** p da u a v (notazione $p: u \leadsto v$) e' una sequenza di nodi $(u_i | 0 \leq i \leq n)$ t.c. $u = u_0$, $v = u_n$ e per ogni $i < n$ la coppia (u_i, u_{i+1}) e' un arco. Il numero naturale n e' il numero di archi del cammino p (n puo' essere $= 0$). p e' un **cammino semplice** sse i nodi di p (e quindi anche gli archi) sono distinti. p e' un **ciclo** sse $p: u \leadsto u$ e p contiene almeno un arco, ed inoltre e' un **ciclo semplice** se i nodi sono distinti, tranne il primo e l'ultimo. v e' **raggiungibile** da u sse esiste un cammino $p: u \leadsto v$.
- una **componente fortemente connessa** di G e' una classe di equivalenza rispetto alla relazione di equivalenza R su V definita da uRv sse u e' raggiungibile da v e viceversa.

Elenchiamo alcune proprieta' di interesse per un grafo orientato G:

- G e' **semplice** sse non contiene cappi.
- G e' **aciclico** (DAG) sse non contiene cicli semplici. Se esiste un ciclo $p: u \leadsto u$, allora esiste un segmento iniziale di p che e' un ciclo semplice. Quindi un grafo e' aciclico sse non contiene cicli. G e' una **foresta** sse e' aciclico ed ogni nodo ha in-degree ≤ 1.
- G e' **fortemente connesso** sse per ogni coppia di nodi (u, v) esiste un cammino $p: u \leadsto v$.

I Grafi orientati possono essere visti come strutture relazionali per la segnatura con una sort ed un predicato binario, quindi si possono definire (nel modo usuale) le nozioni di

- omomorfismo $f: G_1 \to G_2$ sse $f: V_1 \to V_2$ t.c. $(u, v) \in E_1$ implica $(fu, fv) \in E_2$
- isomorfismo $f: G_1 \to G_2$ sse $f: V_1 \to V_2$ bigettiva e t.c. $(u, v) \in E_1$ sse $(fu, fv) \in E_2$
- G_1 sottografo di G_2 sse V_1 sottoinsieme di V_2 e $(u, v) \in E_1$ implica $(u, v) \in E_2$
- G_1 sottografo di G_2 generato da V_1 (con V_1 sottoinsieme di V_2) sse $\forall u, v \in V_1.(u, v) \in E_1 \leftrightarrow (u, v) \in E_2$.

4.1 Grafi: varianti

Un grafo non orientato si puo' vedere come un caso particolare di grafo orientato senza cappi e t.c. $(u, v) \in E$ implica $(v, u) \in E$. Le nozioni introdotto per i grafi orientati si trasferiscono quasi immutati ai grafi non orientati, le principali differenze riguardano

- grado di u = numero di archi entranti in u = numero di archi uscenti da u
- un cammino/ciclo $p: u \leadsto v$ in un grafo non orientato corrisponde ad un cammino/ciclo semplice (nel corrispondente grafo orientato).

Esistono molte variazioni della nozione di grafo orientato (e non orientato):

- Multigrafo (V, E, dom, cod) con $dom, cod: E \to V$, cioe' vi possono essere piu' archi tra due nodi prefissati
- Ipergrafo (V, E, inc) con $inc: E \to V^*$, cioe' un iperarco identifica una sequenza finito di nodi
- Grafo con nodi etichettati (V, E, l) con $l: V \to L$, cioe' ad ogni node u e' assegnata una etichetta $l(u)$
- Grafo con archi etichettati (V, E, l) con $l: E \to L$, cioe' ad ogni arco e e' assegnata una etichetta $l(e)$.
- Grafo ordinato (V, E) se gli archi uscenti da un nodo sono ordinati linearmente.

4.2 Rappresentazione di grafi orientati

Le rappresentazioni piu' comunemente utilizzate per un grafo (orientato) sone quelle a matrice di adiacenza e a liste di adiacenza. Supponiamo fissato l'insieme dei nodi V, allora *astrattamente* un grafo $G = (V, E)$ puo' essere rappresentato

- con un array di insiemi di adiacenza `A: array [V] of "sottoinsiemi di V"`, la relazione di corrispondenza tra grafo e sua rappresentazione e' data da: (u, v) e' un arco sse v e' in $A[u]$.
- con un insieme di coppie ordinate `B: "sottoinsiemi di VxV"`, la relazione di corrispondenza tra grafo e sua rappresentazione e' data da: (u, v) e' un arco sse (u, v) e' in B.

Queste due rappresentazioni astratte possono essere specializzate scegliendo una delle rappresentazioni dei sottoinsiemi (bit-vector, sequenza senza ripetizioni, sequenza ordinata senza ripetizioni).

- se scegliamo di rappresentare i sottoinsiemi con dei bit-vetcors otteniamo la rappresentazione a matrice di adiacenza `M: array [V,V] of boolean`, $M[u, v]$ true sse (u, v) e' un arco.
- se scegliamo di rappresentare i sottoinsiemi con sequenze senza ripetizioni otteniamo la rappresentazione a sequenze di adiacenza (che a loro volta possono essere rappresentate con array, files, o liste con puntatori). Usando le sequenze senza riperizioni si rappresentano di fatto grafi ordinati.

Nel caso delle foreste (viste come grafi orientati semplici e con in-degree ≤ 1), si puo' adottare una rappresentazione piu' compatta `P : array [V] of V+{nil}`, che sfruttare la proprieta' che ogni nodo ha al piu' un padre, la corrispondenza tra foresta e rappresentazione e' data da: u padre di v sse $u = P[v]$. Non tutti gli array P corrispondono ad una foresta, p.e. P potrebbe rappresentare un grafo costituito da un ciclo semplice. Se $n = |V|$, si puo' convenire di enumerare i nodi da 1 ad n (ed identificare nil con 0) in modo che se v_i e' padre di v_j allora $i < j$.

4.3 Topological sorting

Possiamo utilizzare un grafo orientato per rappresentare le dipendenze tra diverse attivita', piu' precisamente una attivita' (o evento) puo' essere identificata con un nodo, e la presenza di un arco $u \to v$ indica che l'attivita' v puo' iniziare solo dopo il completamento dell'attivita' u. In questo contesto si e' interessati a risolvere i seguenti problemi:

- trovare le attivita' che non dipendono dal completamento di altre, ovvero che hanno in-degree $= 0$
- verificare l'assenza di cicli (che renderebbero impossibile l'espletamento di alcune attivita')
- trovare un ordine di esecuzione delle attivita' compatibile con i vincoli imposti dal grafo
- trovare una tempistica ottimale di espletamento delle attivita' (alcune delle quali potrebbero essere eseguite in parallelo). In quest'ultimo caso si potrebbe etichettare il nodo u con il tempo richiesto per svolgere l'attivita', e l'arco (u, v) con il tempo che deve intercorrere tra la fine dell'attivita' u e l'inizio dell'attivita' v.

Per tutti questi problemi e' possibile dare degli algoritmi che lavorano in tempo $O(|V| + |E|)$. Descriveremo gli algoritmi usando la rappresentazione astratta dei grafi come array di insiemi di adiacenza. Prima di procedere alla descrizione degli algoritmi diamo una definizione piu' formale dei problemi:

- calcolo dello in-degree (grado entrante). `input  G grafo` ,
  `output IN:array [V] of Nat` t.c. $In[u] =$ in-degree di u per ogni $u \in V$
- test di aciclicita'. `input  G grafo` ,
  `output nocicle:Bool` t.c. *nocicle* sse G non contiene cicli
- topological sorting (di un grafo aciclico). `input  G grafo` ,
  `output l lista di nodi` t.c. $l = (v_i | i \in |V|)$ senza ripetizioni e $\forall i, j \in |V|. v_i \to v_j \supset i < j$
- tempistica ottimale. `input  G grafo con archi etichettati in Nat` ,
  `output T:array [V] of Nat` t.c. T e' la tempistica ottimale compatibile con il grafo, dove T e' compatibile con il grafo se $\forall (u, v) \in E. T[u] + l(u, v) \leq T[v]$, e T e' migliore di T' se $\forall u \in V. T[u] \leq T'[u]$.

4.3.1 Calcolo dello in-degree

```
input  A:array [V] of "sottoinsiemi di V"
output IN:array [V] of Nat

for v in V do "azzera IN[v]" {inizializzazione di IN}

for u in V do
  for v in A[u] do
    "incrementa IN[v]"
```

Lo out-degree di u e' semplicemente la cardinalita' di $A[u]$, se i sottoinsiemi sono rappresentati con un array allora la cardinalita' la di $A[u]$ si calcola in $O(1)$, e quindi l'out-degree si calcola in $O(|V|)$.

4.3.2 Topological sorting

```
input  A:array [V] of "sottoinsiemi di V"
output l:coda
var    IN:array [V] of Nat {vettore con in-degrees}
       X0:coda/stack {insieme dei nodi con in-degree 0}

"inizializza IN" {usando algoritmo per calcolo dell'in-degree}
X0 <- {u|IN[u]=0}
l  <- empty

while "X0 non vuoto" do
  "rimuovi un elemento u da X0"
  "inserisci u in l"
  for v in A[u] do  {rimuovi u dal grafo, aggiornando IN e X0}
    "decrementa IN[v]"
    if IN[v]=0 then "aggiungi v ad X0"
```

Se al termine del ciclo while l'array IN non ha tutti gli elementi nulli, allora vuol dire che il grafo contiene un ciclo. Le dipendenze tra le celle di uno spreadsheet possono essere rappresentate mediante un grafo ordinato, l'aciclicita' del grafo significa che alle celle e' possibile assegnare un valore calcolato, ed il topological sorting indica in che ordine calcolare il valore delle singole celle (per essere sicuri che i valori di tutte le celle da cui tale valore dipende siano gia' stati calcolati). La correttezza dell'algoritmo si puo' dimostrare con la tecnica dell'invariante, piu' precisamente le seguenti proprieta' valgono prima di eseguire un ciclo dello while

- l e' una lista senza ripetizioni, e $\forall(u,v) \in E$ se $v \in l$ allora u precede v in l
- $IN[u]$ = in-degree di u nel grafo G a cui sono stati tolti tutti gli archi uscenti da nodi in l
- $X0$ e' l'insieme dei nodi u t.c. $IN[u] = 0$ e $u \notin l$.

4.3.3 Tempistica ottimale

L'algoritmo ha la stessa struttura di quello per il topological sorting.

```
input  G:grafo con archi etichettati
output T:array [V] of Nat
var    IN:array [V] of Nat {vettore con in-degrees}
       X0:coda/stack {insieme dei nodi con in-degree 0}

"inizializza IN" {usando algoritmo per calcolo dell'in-degree}
X0 <- {u|IN[u]=0}
for u in V do T[u]<-0

while "X0 non vuoto" do
  "rimuovi un elemento u da X0"
  for v adiacente ad u do {rimuovi u dal grafo, aggiornando IN e X0}
```

```
    T[v]<-max(T[v],T[u]+l(u,v))
    "decrementa IN[v]"
    if IN[v]=0 then "aggiungi v ad X0"
```

4.4 Visite di un grafo orientato

Fissato un nodo r di un grafo orientato $G = (V, E)$ diciamo che una sequenza $S = ((u_i, v_i)|1 \leq i \leq k)$ di archi e' una **visita** di G a partire da r sse

- l' i-esimo arco (u_i, v_i) della sequenza e' t.c. $u_i \in V_{i-1}$ mentre $v_i \notin V_{i-1}$, dove V_i e' definito per induzione su $0 \leq i \leq k$: $V_0 = \{r\}$ e $V_{i+1} = V_i \cup \{v_{i+1}\}$
- non esiste un arco (u, v) t.c. $u \in V_k$ ma $v \notin V_k$.

Ogni visita soddisfa le seguenti proprieta':

- V_k e' l'insieme dei nodi raggiungibili da r.

  Dimostrazione per assurdo. Se v e' raggiungibile da r ma non e' in V_k, si consideri il primo nodo u in un cammino da r a v che non sta in V_k, si ottenga una contraddizione con la seconda condizione nella definizione di visita.
- il grafo $G_i = (V_i, E_i)$ con $0 \leq i \leq k$, dove E_i e' l'insieme dei primi i archi in S, e' un albero con radice r.

Il seguente algoritmo fornisce uno schema generale per produrre una visita di $G = (V, E)$ a partire da r.

```
input  A:array [V] of "sottoinsiemi di V" {liste di adiacenza per G}
       r:V {nodo di partenza}
output S:coda di archi
var    U:sottoinsieme di V {dei nodi visitati}
       F:sottoinsieme di V {dei nodi visitati frontiera}

U <- F <- {r}
S <- empty

while "F non vuoto" do
  u <- "un elemento di F"
  if A[u] incluso in U
    then "rimuovi u da F"
    else v <- "un elemento di A[u]-U"
         "inserisci v in U ed F"
         "inserisci (u,v) in coda a S"
```

Le visite si possono sfruttare per risolvere una varieta' di problemi sui grafi: raggiungibilita', test di connettivita' (forte), calcolo delle componenti (fortemente) connesse. Per esempio, per stabilire se un grafo orientato G e' fortemente connesso basta prendere un nodo r e verificare se tutti i nodi sono raggiungibili da r sia nel grafo G che nel suo *duale* (ottenuto invertendo la direzione degli archi). Se S e' una visita, introduciamo la seguente notazione

- v_i e' l'i-esimo nodo visitato, e si conviene di indicare r con v_0. Quindi l'i-esimo arco di S e' della forma (u_i, v_i).
- p_i e' la posizione nella lista dei nodi visitati di u_i, cioe' il padre di v_i nell'albero associato ad S, ovvero $u_i = v_{p_i}$. Quindi p_0 e' indefinito, e $p_i < i$ se $i > 0$.
- V_i e' l'insieme $\{v_0, ..., v_i\}$.

4.4.1 Algoritmi di visita BFS e DFS

Vi sono due visite che si possono effettuare in modo efficiente, cioe' in $O(|V| + |E|)$, la visita in ampiezza (detta BFS) e quella in profondita' (detta DFS). Nella visita BFS si implementa F come una coda

```
var U:sottoinsieme di V {dei nodi visitati}
    F:coda di V {dei nodi visitati frontiera}

U <- F <- {r}
```

```
S <- empty

while "F non vuota" do
  "estrai il primo elemento u di F"
  for v in A[u] do
    if "v non in U"
      then "inserisci v in U ed F"
           "inserisci (u,v) in coda a S"
```

Nella visita DFS si implementa F come uno stack, o piu' semplicemente si usa una procedura ricorsiva

```
var U:sottoinsieme di V {dei nodi visitati}
U <- empty
S <- empty
DFS(r)
dove
DFS(u:V)
"inserisci u in U"
for v in A[u] do
  if "v not in U"
    then "inserisci (u,v) in coda a S"
         DFS(v)
```

Vediamo come si adatta l'algoritmo DFS per determinare le componenti connesse di un grafo NON ORIENTATO.

```
input  A:array [V] of sottoinsiemi di V
output c:array [V] of 0..|V| {c[v] numero della componente connessa}
var    k: 0..|V| {numero componenti connesse}
       U:sottoinsieme di V {dei nodi visitati}

U <- empty
k <- 0
for v in V do c[v] <- 0

while "V-U non vuoto" do
  k <- k+1
  r <- "un elemento di V-U"
* invariante:
* 0 < c[v] <= k se v in U, c[v] = 0 altrimenti
* {v|c[v]=i} componente connessa se 0 < i < k
* c[v]=k sse v in U e v nella stessa componente connessa di r
  DFS'(r)

DFS'(u:V)
"inserisci u in U"
c[u] <- k
for v in A[u] do
  if "v not in U" then DFS'(v)
```

4.4.2 Correttezza e terminazione degli algoritmi di visita

In questa sezione formaliziamo gli algoritmi di visita mediante automi deterministici, dimostrandone la correttezza parziale (con la tecnica dell'invariante) e la terminazione. Per ottenere degli automi deterministici occorre risolvere il non-determinismo in alcuni passi dell'algoritmo di visita. Il problema della visita di un grafo possiamo formularlo nel modo seguente (supponiamo che l'insieme V dei nodi sia fissato)

- l'insieme I degli input e' costituito dalle coppie (E, r) con $E \subseteq V \times V$ e $r \in V$
- l'insieme O degli output e' $(V \times V)^*$
- il problema $P \subseteq I \times O$ e' $(E, r)\ P\ S$ sse S e' una visita di $G = (V, E)$ a partire da r.

L'algoritmo generale di visita si puo' formalizzare con il seguente automa

- gli stati interni sono della forma (E, S, U, F) con $E \subseteq V \times V$, $S \in (V \times V)^*$ e $U, F \subseteq V$. U e' l'insieme dei nodi visitati, mentre F e' l'insieme dei potenziali nodi frontiera.
- la relazione di transizione e' data da
  - $(E, r) \longrightarrow (E, nil, \{r\}, \{r\})$ inizializzazione
  - $(E, S, U, \emptyset) \longrightarrow S$ output
  - $(E, S, U, F) \longrightarrow (E, S, U, F - u)$ se u *primo elemento* di F e $A[u] \subseteq U$, dove $A[u] = \{v | (u, v) \in E\}$
  - $(E, S, U, F) \longrightarrow (E, S\colon (u, v), U + v, F \cup v)$ se u *primo elemento* di F e v *primo elemento* di $\{v \in A[u] | v \notin U\}$

Per dimostrare la correttezza parziale dell'algoritmo utilizziamo il l'invariante costituito dalle coppie input (E, r) stato interno (E, S, U, F) t.c.

- S soddisfa la prima proprieta' per essere una visita di G a partire, cioe' $S = ((u_i, v_i) \in E | 1 \leq i \leq k)$ e $\forall i \in [1..k]. u_i \in V_{i-1} \wedge v_i \notin V_{i-1}$, dove $V_i = \{r\} \cup \{v_j | 1 \leq j \leq i\}$
- $U = V_k$
- $\{u \in U | A[u] \not\subseteq U\} \subseteq F \subseteq U$

Per dimostrare la terminazione dell'algoritmo definiamo una funzione $p(E, S, U, F) = 2 * |V - U| + |F|$, dove $|F|$ e' la cardinalita' dell'insieme F. Occorre quindi dimostrare che ogni transizione tra stati interni fa' diminuire il valore di p.

L'algoritmo di visita BFS si puo' formalizzare con il seguente automa

- gli stati interni sono della forma (E, S, U, F) con $E \subseteq V \times V$, $S \in (V \times V)^*$, $U \subseteq V$ e $F \in V^*$ U e' l'insieme dei nodi visitati, mentre F e' la coda dei potenziali nodi frontiera.
- la relazione di transizione e' data da
  - $(E, r) \longrightarrow (E, nil, \{r\}, r)$ inizializzazione
  - $(E, S, U, nil) \longrightarrow S$ output
  - $(E, S, U, u\colon F) \longrightarrow (E, S@((u, v) | v \in A'[u]), U \cup A'[u], F \cdot A'[u])$, dove $A'[u]$ e' la lista/insieme $[v \in A[u] | v \notin U]$

Per dimostrare la correttezza parziale dell'algoritmo utilizziamo lo *stesso* invariante usato per la correttezza parziale dell'algoritmo generale di visita, identificando la lista F con l'insieme dei suoi elementi Per dimostrare la terminazione dell'algoritmo definiamo la funzione $p(E, S, U, F) = |V - U| + |F|$, dove $|F|$ e' la lunghezza della lista F. Occorre quindi dimostrare che ogni transizione tra stati interni fa' diminuire il valore di p.

Contents