Esercitazione 6
Algorithmi e Strutture Dati (Informatica)
A.A 2015/2016

Tong Liu

April 14, 2016

Elementi Fondamentali

Rappresentazione

$n = |V|$ numero di vertici (nodi) $m = |E|$ numero di archi

Matrice di adiacenza:
- Spazio richiesto $O(n^2)$
- Verificare se il vertice u è adiacente a v richiede tempo $O(1)$
- Elencare tutti gli archi costa $O(n^2)$

Liste di adiacenza:
- Spazio richiesto $O(n+m)$
- Verificare se il vertice u è adiacente a v richiede tempo $O(n)$
- Elencare tutti gli archi costa $O(n+m)$

Lista di termini

grado(degree), grado entrante(incoming degree);grado uscente, cammino, catena, ciclo, circuito, aciclico, DAG(Directed Acyclic Graph), grafo completo(complete graph), albero libero - una catena per ogni coppia di nodi, albero radicato - un vertice è designato come radice, foresta,

Algoritmi base

DFS Pseudocodice 1, BFS Pseudocodice 2

Algorithm 1 Algoritmo BFS

```
1:  procedure BFS(Graph G, Node r)
2:      QUEUE S ← Queue()
3:      S.enqueue(r)
4:      boolean[] visitato ← new boolean[1...G.n]
5:      for all u ∈ G.V() do
6:          visitato[u] ← false
7:      end for
8:      visitato[r] ← true
9:      while not S.isEmpty() do
10:         NODE u ← S.dequeue()
11:         {... operazioni relative a u....}
12:         for all v ∈ G.adj(u) do
13:             {... operazioni relative ad arco (u, v) ...}
14:             if not visitato[v] then
15:                 visitato[v] = true
16:                 S.enqueue(v)
17:             end if
18:         end for
19:     end while
20: end procedure
```

Algorithm 2 Algoritmo DFS

```
1:  procedure DFS(Graph G, Node v, boolean[] visitato)
2:      visitato[u] ← true
3:      {... operazioni relative a u....}
4:      for all v ∈ G.adj(u) do
5:          {... operazioni relative ad arco (u, v) ...}
6:          if not visitato[v] then
7:              DFS(G, v,visitato)
8:          end if
9:      end for
10: end procedure
```

Esercizi

Esercizio Facebook Amici di distanza 2

Un social network come Facebook può essere rappresentato mediante un grafo non orientato $G = (V, E)$, in cui ogni nodo rappresenta un utente, ed esiste un arco tra i nodi u e v se e solo se gli utenti rappresentati da u e v sono "amici"; assumiamo che se u è "amico" di v, allora anche v è "amico" di u (quindi G è un grafo non orientato). Nota: il grafo G potrebbe non risultare connesso.
1. Scrivere un algoritmo efficiente che, dato in input un grafo non orientato $G = (V, E)$ e un nodo $v \in V$, restituisce il numero di "amici di amici" di v, ossia restituisce il numero di nodi che si trovano a distanza 2 da v. Non è consentito usare variabili globali.
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.

Svolgimento

Questo problema si può risolvere utilizzando una versione adattata della visita in ampiezza (BFS). È anche possibile realizzare una soluzione che non fa uso esplicito di una coda. La soluzione è descritta in Pseudocodice 3

Algorithm 3 Algoritmo Facebook Amici di distanza 2

```
1:  procedure CONTAAMICIDISTANZA2(Graph G, Node v)
2:      Integer c ← 0                                  ▷ Numero di vicini di secondo livello
3:      for all u ∈ G.V() do
4:          u.visited ← false
5:      end for
6:      v.visited ← true
7:      for all u ∈ G.adj(v) do                        ▷ Marca amici di primo livello
8:          u.visited ← true
9:      end for
10:     for all u ∈ G.adj(v) do
11:         for all w ∈ G.adj(u) do                    ▷ Amici di secondo livello
12:             if w.visited = false then
13:                 w.visited = true
14:                 c ← c + 1
15:             end if
16:         end for
17:     end for
18:     return c
19: end procedure
```

Esercizio Nodi alla Massima Distanza

[Libro 9.9] Dato un input un grafo orientato $G = (V, E)$ rappresentato tramite liste di adiacenza e un nodo $r \in V$, restituire il numero dei nodi in G raggiungibili da r che si trovano alla massima distanza da r. Analizzare la complessità.

Svolgimento

È sufficiente adattare BFS con un contatore associato alla coda, che memorizza il numero di elementi alla massima distanza trovata fino ad un certo punto. Complessità $O(m+n)$. Pseudocodice 4.

Algorithm 4 Algoritmo Numero di Nodi alla Massima Distanza con BFS

```
1:  procedure MAXDISTANCE(GRAPH G, NODE r)
2:      d ← new integer[1...G.n]
3:      Integer max ← 0
4:      Integer count ← 0
5:      Queue S ← Queue()
6:      S.enqueue(r)
7:      for all u ∈ G.V() do
8:          d[u] ← ∞
9:      end for
10:     d[r] ← 0
11:     while not S.isEmpty() do
12:         NODE u ← S.dequeue()
13:         for all v ∈ G.adj(u) do
14:             if dist[v] = ∞ then
15:                 d[v] ← d[u] + 1
16:                 if d[v] > max then
17:                     max ← d[v]
18:                     count ← 1
19:                 else if d[v] =max then
20:                     count ← count + 1
21:                 end if
22:                 S.enqueue(v)
23:             end if
24:         end for
25:     end while
26:     Return count
27: end procedure
```

Esercizio Diametro Grafo

[Libro 9.11] Il diametro di un grafo è la lunghezza del 'più lungo cammino breve', ovvero la più grande lunghezza del cammino minimo (in termini di numero di archi) fra tutte le coppie di nodi. Progettare un algoritmo che misuri il diametro di un grafo e valutare la sua complessità.

Svolgimento

Si effettuano n visite in ampiezza, una a partire da ogni nodo, e si memorizza il massimo valore di distanza trovato in una visita. Complessità $O(n^2 + mn)$. Pseudocodice 5.

Algorithm 5 Algoritmo Diametro Grafo con BFS visita in ampiezza

```
1:  procedure DIAMETRO(GRAPH G)
2:      Integer max ← 0
3:      for all r ∈ G.V() do
4:          Queue S ← Queue()
5:          S.enqueue(r)
6:          integer[ ] dist ← new integer[1...G.n]
7:          for all u ∈ G.V() do
8:              dist[u] ← - 1
9:          end for
10:         dist[r] ← 0
11:         while not S.isEmpty() do
12:             NODE u ← S.dequeue()
13:             for all v ∈ G.adj(u) do
14:                 if dist[v] < 0 then
15:                     dist[v] ← dist[u] + 1
16:                     if dist[v] > max then
17:                         max ← dist[v]
18:                     end if
19:                     S.enqueue(v)
20:                 end if
21:             end for
22:         end while
23:     end for
24: end procedure
```

Esercizio Componenti Connessi

[Esame 19/06/2015] Scrivere in pseudocodice un algoritmo che, dato un grafo non orientato G, conti il numero delle sue componenti conesse che sono anche alberi. Discutere correttezza e complessità dell'algoritmo proposto.

Svolgimento

Per risolvere questo esercizio, è sufficiente modificare l'algoritmo che visita le componenti connesse, in modo da verificare, per ogni componente, se la componente contiene cicli oppure no. Bisogna prestare attenzione al fatto che tutti gli archi sono sdoppiati, quindi quando si visita un nodo v per la prima volta arrivando dal nodo u bisogna evitare di considerare l'arco $[v,u]$. La complessità è $O(m+n)$. Pseudocodice 6.

Algorithm 6 Algoritmo per Componenti Connessi e Alberi

```
1:  procedure COUNTTREES(Graph G)
2:      boolean[] visitato ← new boolean [1 ... G.n]
3:      for all u ∈ G.V() do
4:          visitato[u] ← false
5:      end for
6:      Integer count ← 0
7:      for all u ∈ G.V() do
8:          if not visitato[u] then
9:              if countTreesDFS(G, u, nil, visitato) then
10:                 count ← count + 1
11:             end if
12:         end if
13:     end for
14:     return count
15: end procedure
16: procedure COUNTTREESDFS(GRAPH G, NODE u, NODE p, boolean[] visitato)
17:     boolean isAcyclic ← true
18:     visitato[u] ← true
19:     for all v ∈ G.adj(u) \ p do
20:         if not visitato[v] then
21:             if not countTreesDFS(G, v, u, visitato) then
22:                 isAcyclic ← false
23:             end if
24:         else
25:             isAcyclic ← false
26:         end if
27:     end for
28:     return isAcyclic
29: end procedure
```

Esercizio Ordinamento Topologico

[Libro 9.7] Siano dati n programmi P_i che richiedono tempo T_i per essere eseguiti. Tra i programmi è definita una relazione di precedenza $P_i < P_j$ che indica che l'esecuzione di P_i

deve essere terminata prima dell'inizio di P_j. Si fornisca una procedura per determinare in tempo polinomiale l'istante in cui iniziare l'esecuzione di ciascun programma in modo da rispettare tutti i vincoli e terminare l'esecuzione il prima possibile.

Svolgimento

Si possono rappresentare i programmi come nodi di un grafo orientato aciclico dove gli archi rappresentano le relazioni di precedenza, cioè se $P_i < P_k$ allora (i, j) è un arco del grafo. Gli istanti di partenza si calcolano scandendo la sequenza dei nodi ordinati topologicamente, e assegnado all'istante di partenza di ogni nodo in valore che è determinato dalla somma dei tempi di esecuzione dei nodi precedenti. Oltre al vettore *ordine*, si utilizzano i vettori *durata* e *partenza* che contengono, rispettivamente, i tempi di esecuzione di ogni programma e gli istanti di partenza di ciascun nodo. La complessità dell'algoritmo è $O(n+m)$. Pseudocodice 7

Algorithm 7 Algoritmo Ordinamento dipendenze programmi

```
1:  procedure SCHEDULING(GRAPH G, Integer[] durata)
2:      ordine ← new integer[1 ... G.n ]
3:      topSort(G,ordine)
4:      partenza ← new integer[1 ... G.n ]
5:      partenza[ordine[1]] ← 0
6:      for i ← 2 to G.n do
7:          Integer j ← ordine[i]
8:          partenza[j] ← partenza[j] + durata[j]
9:      end for
10: end procedure
```