Grafi

Una coppia di insiemi G = <V, A>

V = vertici

A = VxV => archi

G = {1,2,3}

A = {(1,2), (2,3)} (seguente, destinazione)

A ={(1,2), (2,1), (2,3), (3,2)}

Grafo indiretto

V = {vertici}

A = {{a,b} t.c. a,b appartiene a V}

V = {1,2,3}

A = {(1,2), (2,3)}

Popolarità di un nodo

Per capire la popolarità di un nodo si può controllare

Archi_entranti(v) = {(u,v) | u in V e (u,v) in A}

Numero massimo di archi per n nodi

Grafo diretto:

- con n nodi
- archi n^2

Grafo indiretto:

- con n nodi
- archi ((n^2)/2)+(n/2) = n(n+1)/2

Implementare un grafo

I nodi un tramite Array. Questa rappresentazione tramite Array si chiama lista di adiacenza e vengono descritti i nodi raggiungibili tramite una lista puntata di nodi.

Oppure si può utilizzare anche una mappa (array 2D)

Un nodo è raggiungibile a partire da un altro se esiste un percorso che lo collega.

Costo Spaziale

Lista di adiacenza

Lista di tutti gli elementi presenti all'interno del grafo con i relativi archi che collegano il nodo agli altri elementi del grafo.

G = (V, E) modello del grafo.

Nel caso pessimo se ogni nodo si collega ad ogni altro nodo |E| = O(n^2), in generale minore o uguale a M^2.

Lo spazio sarà la sommatoria delle liste in ogni elemento della struttura dati, quindi il numero di archi che ho nel mio grafo. L'essenzaile per poter descrivere il mio grafo.

Matrice bidimensionale

Spazio: quadratico delle celle della matrice |V|^2, si spreca tantissimo spazio. La lista di adiacenza è ben più efficiente quando si parla di grafi con tanti nodi.

Visite

Procedura che porta ad identificare tutti i nodi seguendo gli archi.

Possiamo fare quello che abbiamo fatto con gli alberi sul grafo? Si, ma la situazione è più complessa.

- Possono esserci nodi che non sono legati a nessun altro nodo.
- Possono esserci dei "DAG" (directed acyclic graph), in cui dei nodi vengono visitati più volte.
- Possono esseri dei cicli (caso generale), il problema principale.

I DAG

In un DAG: più radici e un nodo anche più di un genitore. Modello più complesso di un albero, ma non generale come quello di un grafo diretto.

Con i DAG la visita può avere un costo **esponenziale**.

Bisogna affrontare il problema dei cicli e delle visite

Situazione in cui si entra in un loop e non se ne esce più.

Caso minimo 2 nodi con 2 archi che li collegano tra di loro, oppure 1 solo nodo che è collegato tramite un arco a se stesso.

Dobbiamo cambiare qualcosa nell'algoritmo di visita. Dobbiamo trovare una visita lineare, quindi senza cadere in cicli ed esplodere esponenzialmente.

Visita Depth First search (DFS)

Rec (n){

```
n visitato?

si -> return; n.visitato = 1;

for (archi uscenti)

    rec (n->m)
```

}

Sto aggiungedo un flag, quindi aumento lo spazio occupato di n, ma posso fare visite molto più veloci (n.visitato)

Visita di tutti i successivi di A: Breadth first search (BFS) (Ricerca in ampiezza)

Iterativa+Coda

Visita (A){

```
Inserisco in coda A

while (E <- Coda){
    successivi di E -> Coda
    Visito E
}
```

}

La coda ci permette di espandere la visita in aneli concentrici di distanza nel nostro grafo, pemettendoci anche di non creare cicli.

Ci permette di trovare il più corto percorso tra due nodi.

Weither a graph (Grafo con i pesi)

Aggiungo l'informazione del peso di ogni arco. Per esempio in una strada so potrebbe aggiungere la velocità consentita o il tempo di percorrenza medio.

A ---> B (A, B, weight)

La Breadth first search non funziona con i pesi perchè non vengono considerati.

Percorso più corto?

- Elenco il percorsi
- Costo percorso
- Minimo?

Funziona? Si, ma è lentissimo, si va di costo esponenziale.

Nuovo nodo per risolvere il problema:

- Costo per raggiungere il nodo (init = infinito)
- precedente (init = null)

La vista in stile BFS però mi crea un problema, devo espandere tutti i percorsi per poi rimetterli a posto (per esempio dopo un miglioramento di un percorso), bisogna aggiungere qualcos altro per poter avere una vista efficace con i pesi. Praticamente con un grafo complesso potrei dover risistemare la visita anche un numero esponenziale di volte.

- Devo espandere prima gli archi con il costo minimo a livello di peso.

Queto tipo di visita viene chiamata **L'algoritmo di Dijkstra** link Wikipedia, oppure anche **shortest path**

Ci sono anche delle visite con ordinamento dei nodi. Ordinamento topologico

Ordinamento Topologico

Uno degli algoritmi classici per ordinare topologicamente un grafo è basato sul concetto di visita in profondità (ricerca depth-first). L'algoritmo di ordinamento topologico effettua una visita in profondità del grafo (a partire da ognuno dei nodi senza archi entranti) e, terminata la visita di ogni vertice, lo inserisce in testa ad una lista concatenata L. Al termine dell'esecuzione, questa lista conterrà i nodi ordinati. In pseudocodice:

```
L ← Lista vuota (conterrà i nodi ordinati)
S ← Insieme di tutti i nodi senza archi entranti
for each nodo n in S do
    visit(n)
return L
```

dove la procedura visit è definita come

```
function visit(nodo n)
    if n non è ancora stato visitato then
        marca n come visitato
        for each nodo m con un arco da n a m do
            visit(m)
        aggiungi n a L
```

Componenti fortemente connesse

Wiki Una componente fortemente connessa di un grafo diretto G è un sottografo massimale di G in cui esiste un cammino orientato tra ogni coppia di nodi ad esso appartenenti.

Abbiamo un grafo diretto (se il grafo non è diretto non c'è il "fortemente"), quindi non è detto che tutti i percorsi siano invertibili.

Ipotesi grafo con tanti nodi tra cui A e B:

- Posso andare dal nodo A al nodo B? Si
- Posso andare dal nodo B al nodo A? Si

Allora A e B saranno nella stessa componente fortemente connessa.

Presa una qualunque coppia di nodi devo poter andare da l'uno al'altro e sapere qualsi sia la strada di ritorno.

Algoritmo disponibili:

- Tarjan's strongly connected components algorithm Wikipedia
- Kosaraju's algorithm Wikipedia
- Path-based strong component algorithm Wikipedia

Algoritmo di Tarjan

The algorithm takes a directed graph as input, and produces a partition of the graph's vertices into the graph's strongly connected components. Each vertex of the graph appears in exactly one of the strongly connected components. Any vertex that is not on a directed cycle forms a strongly connected component all by itself: for example, a vertex whose in-degree or out-degree is 0, or any vertex of an acyclic graph.

Idea alla base dell'algoritmo:

The basic idea of the algorithm is this: a depth-first search (DFS) begins from an arbitrary start node (and subsequent depth-first searches are conducted on any nodes that have not yet been found). As usual with depth-first search, the search visits every node of the graph exactly once, declining to revisit any node that has already been visited. Thus, the collection of search trees is a spanning forest of the graph. The strongly connected components will be recovered as certain subtrees of this forest. The roots of these subtrees are called the "roots" of the strongly connected components. Any node of a strongly connected component might serve as a root, if it happens to be the first node of a component that is discovered by search.

Quale implementazione andare ad usare? Esiste la versione a stack e quella a Bookkeeping

Pseudo codice:

```
algorithm tarjan is
    input: graph G = (V, E)
    output: set of strongly connected components (sets of vertices)

    index := 0
    S := empty stack
    for each v in V do
        if v.index is undefined then
            strongconnect(v)
        end if
    end for
```

```
    function strongconnect(v)
        // Set the depth index for v to the smallest unused index
        v.index := index
        v.lowlink := index
        index := index + 1
        S.push(v)
        v.onStack := true

        // Consider successors of v
        for each (v, w) in E do
            if w.index is undefined then
                // Successor w has not yet been visited; recurse on it
                strongconnect(w)
                v.lowlink := min(v.lowlink, w.lowlink)
            else if w.onStack then
                // Successor w is in stack S and hence in the current SCC
                // If w is not on stack, then (v, w) is an edge pointing
to an SCC already found and must be ignored
                // Note: The next line may look odd - but is correct.
                // It says w.index not w.lowlink; that is deliberate and
from the original paper
                v.lowlink := min(v.lowlink, w.index)
            end if
        end for

        // If v is a root node, pop the stack and generate an SCC
        if v.lowlink = v.index then
            start a new strongly connected component
            repeat
                w := S.pop()
                w.onStack := false
                add w to current strongly connected component
            while w ≠ v
            output the current strongly connected component
        end if
    end function
```

È un codice ricorsivo? si c'è una chiamata strongconnect(v)

c'è DFS? Si: for each (v, w) in E do ...

Abbiamo trovato un modo per segnare se ci sono alrchi di rotorno, cicli, componenti connesse.