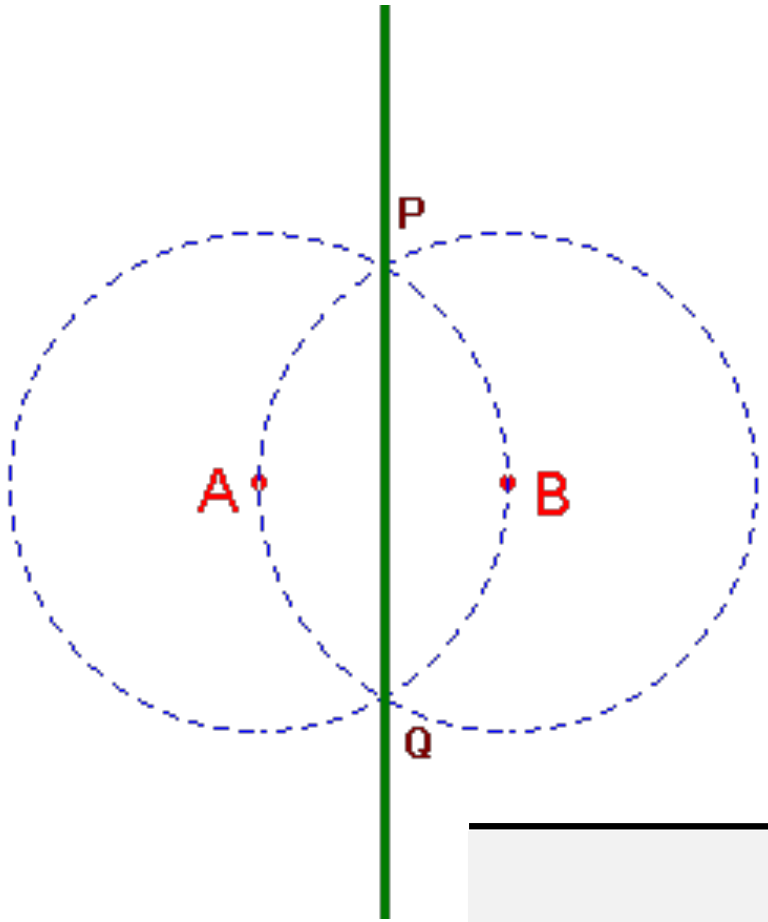
Pot-pourri finale sui grafi

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **26(b)** **[9/1/24]**

SAPIENZA
UNIVERSITÀ DI ROMA



Nodi equidistanti da due nodi fissati

Il problema e soluzioni "ignoranti"

Problema: Progettare un algoritmo che dato un grafo $G = (V, E)$, e due nodi $s, t \in V$, calcola l'insieme dei nodi in V equidistanti da s e da t.

È sempre utile dare una definizione formale. Si tratta di calcolare l'insieme di nodi E definito come:

$$E = \{ v \in V \mid d(s, v) = d(t, v) \}$$

Inoltre, occorre pensare a come **rappresentare la soluzione**, ecco due possibilità: **1**) una **lista di nodi**; **2**) il **vettore caratteristico** di E, cioè un vettore e indicizzato sui nodi, tale che $e[v]$ = TRUE sse $v \in E$.

La definizione si può facilmente trasformare in un algoritmo "ignorante" (nei **riquadri verdi**, la soluzione con **lista**):

```
def eqDist(G, s, t):
    e = allFalse(|G.V|)        L = NULL
    forall v ∈ V:
        if dist(v, s) == dist(v, t):
            e[v] = True        L = cons(v, L)
    return e                   return L
```

La funzione **`dist`** è semplicemente una BFS di complessità $\mathcal{O}(m+n)$ [mi posso fermare prima] e quindi l'intero algoritmo ha complessità $\mathcal{O}(n(m+n))$.

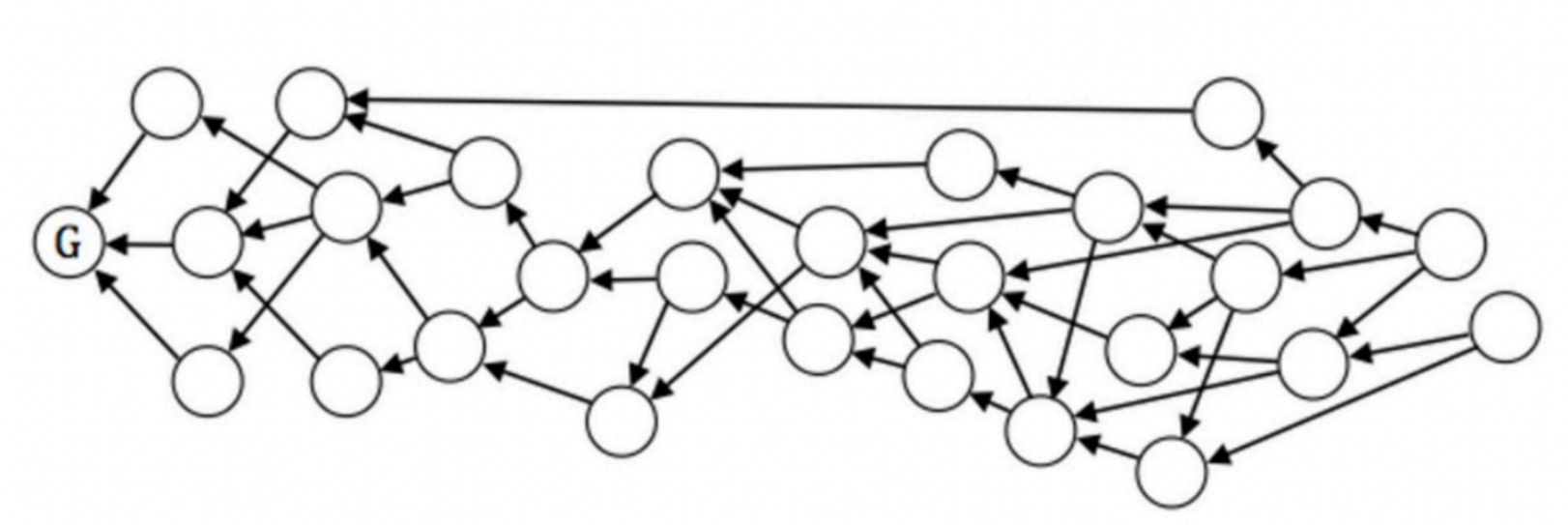
Migliorìe e algoritmo smart

Si potrebbe pensare di fare un'unica BFS per valutare se v sia equidistante da s e t: ciò accade quando s e t stanno sullo stesso livello della BFS radicata in v: probabilmente si **migliora l'efficienza** (al prezzo di una **procedura più complicata**), ma **non si migliora la complessità asintotica**.

Ma ispirandosi alla costruzione geometrica con riga e compasso e ricordando che una singola BFS può calcolare la distanza da un nodo a tutti gli altri, si può ottenere una **procedura molto più efficiente** **che fa solo due BFS**, **una radicata in *s*** e **una in *t*** e poi **confronta i due vettori** di distanze calcolati.

```
def eqDistSmart(G, s, t):
    e = allFalse(|G.V|)
    ds = allDist(G, s)
    dt = allDist(G, t)
    forall v ∈ V:
        if dt[v] == ds[v]:
            e[v] = True
    return e
```

La funzione **`allDist`** è ancora una BFS (completa) di complessità $\Theta(m+n)$, ne faccio solo 2, e il confronto sui vettori è chiaramente $\Theta(n)$ per una complessità totale di $\boldsymbol{\Theta(m+n)}$.

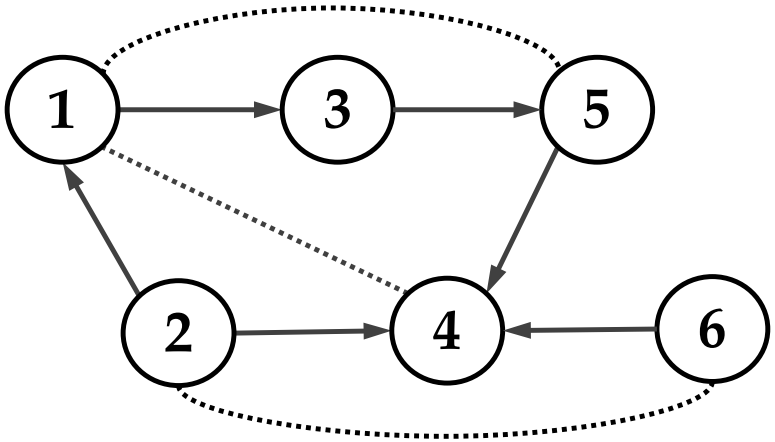


Orientare archi per ottenere un DAG

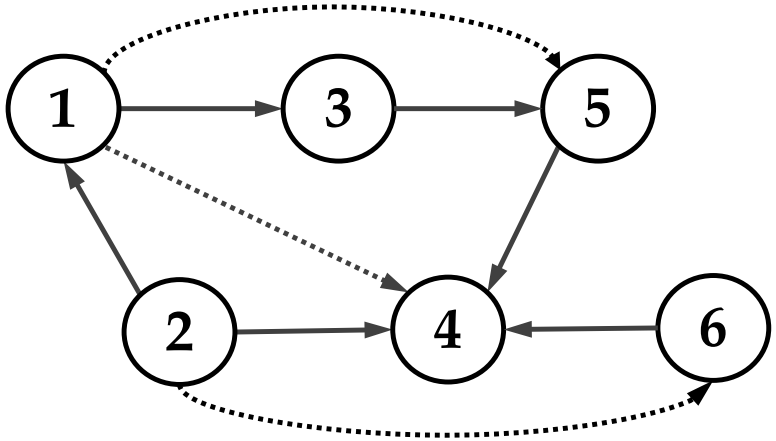
Esercizio 10.4 (IV esonero, 7/6/22)

Problema: Dato un grafo diretto aciclico $G = (V, E)$ e un insieme di archi non orientati $E' \subseteq V \times V$, descrivere un algoritmo per orientare gli archi in E' in modo che possano essere aggiunti a G senza generare cicli.

Come si possono orientare gli archi di un grafo non orientato per ottenere un DAG?



Esempio di DAG e archi da aggiungere (tratteggiati)



Esempio di orientamento degli archi aggiunti tratteggiati

Orientamento di archi

Siccome ***G* è un DAG**, ammette un **ordine topologico $\leq_{TS}$**, e quindi oriento tutti gli archi di E' in accordo con $\leq_{TS}$. L'idea è corretta perché **$\leq_{TS}$** è un **ordine topologico** anche **per G***, che quindi è un DAG.

La finezza era: **come calcolare $\leq_{TS}$ in $\theta(1)$?** Ovviamente la lista ordinata nell'ordine topologico aiutava poco a questo scopo…

Pensando all'algoritmo che iterativamente sceglie nodi con grado entrante 0, si può costruire un vettore **TS** indicizzato sui nodi che registra il **tempo di scelta** di un nodo. Quindi **$u \leq_{TS} v$ sse TS[u] <TS[v]**.

Pensando all'algoritmo che ordina in senso inverso ai tempi di uscita di una DFS… beh, **bastano i tempi di uscita**! In quel caso **$u \leq_{TS} v$ se e solo se $out[u] > out[v]$**.

Chiamando $p = |E'|$ il tutto costa $\theta(n+m+p)$.

$\theta(n+m)$

```
def orienta(G, E′):
    ≤TS = topologicalSort(G)
    forall (u, v) ∈ E′:
        if u ≤TS v: E = E ∪ { u → v }
        else: E = E ∪ { v → u }
    return G
```

$\theta(p)$

Piccole finezze

Quando è possibile orientare un arco **in entrambe le direzioni**?

Quando **non c'è** un **cammino** da u **a** v, né **da** v **a** u. Detto in altri termini, quando c'è sia un ordinamento topologico $\leq_T$ tale che $u \leq_T v$, sia un ordinamento topologico $\leq_{T'}$ tale che $v \leq_{T'} u$.

Infine: **come orientare un generico grafo non orientato** in modo da non creare cicli? Una **visita, certo** (sia BFS che DFS, ma in entrambi i casi occorre **fare attenzione agli archi non dell'albero** di visita).

Però **è sufficiente scegliere un ordinamento topologico arbitrario**, ad esempio quello indotto dalla codifica dei nodi con numeri interi!

```
def orienta(G):
    forall (u, v) ∈ E':
        if u ≤ v:  E = E ∪ { u → v }
        else: E = E ∪ { v → u }
    return G
```

$\Theta(m)$

verifica dei cammini minimi

Esercizio 9.1

➧ **Problema**: Supponete di avere un grafo G con pesi sugli archi w e una soluzione dei cammini minimi da sorgente unica s sottoforma di vettore $c[u]$ dei costi dei cammini per arrivare a u e vettore dei padri p.

Fornire un algoritmo per **verificare che la soluzione sia corretta**.

Un modo per verificarla è la seguente:

1. verificare che il **vettore dei padri p** e **vettori dei costi c sono "fedeli"**, quindi dev'essere $c(p[u]) + w(u, v) = c(v)$ e, ovviamente, dev'esserci l'arco (u, v) in G.
2. verificare che **non ci sono possibilità di "migliorare"** i costi seguendo altri cammini. Un modo è **fare un "giro di Bellman-Ford"** e verificare che `relax` non modifica nessun costo minimo.

Il costo di **1**. è $\Theta(n)$, mentre il costo di **2.** necessita di verificare tutti gli archi.

Dipende dalla rappresentazione, può essere $\Theta(m)$ [a lista di archi], $\Theta(n+m)$ [con liste di adiacenza], $\Theta(n^2)$ [con matrice di adiacenza]

That's all Folks!

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Esercizi lez. **26(b)** **[9/1/24]**

SAPIENZA
UNIVERSITÀ DI ROMA