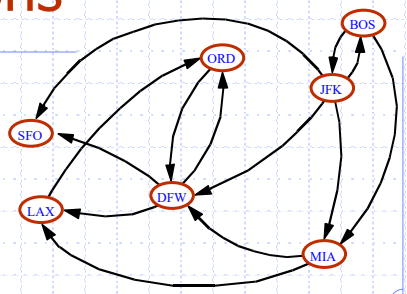
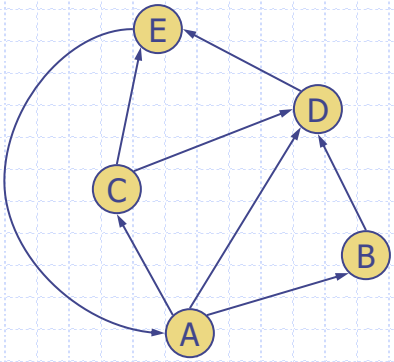
Directed Graphs



1

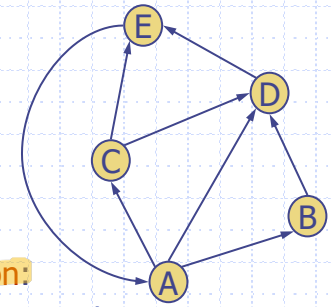
Digraphs

- A digraph is a graph whose edges are all directed
  - Short for "directed graph"
- Applications
  - one-way streets
  - flights
  - task scheduling



2

Digraph Properties

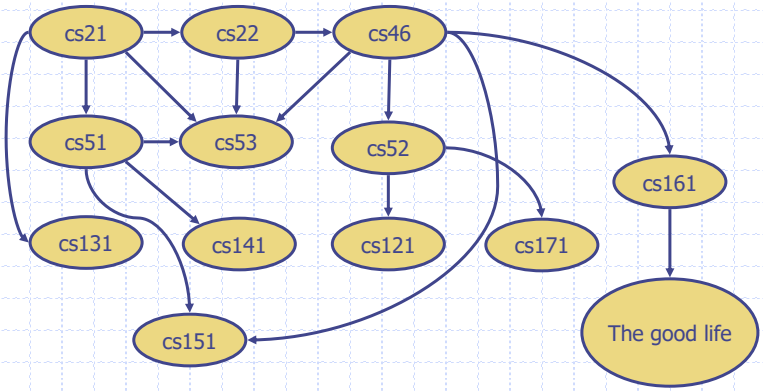


- A graph G=(V,E) such that
  - Each edge goes in one direction:
  - Edge (a,b) goes from a to b, but not b to a
- If G is simple, $m \leq n \cdot (n-1)$ No arcs doppi e self loop
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

3

Digraph Application

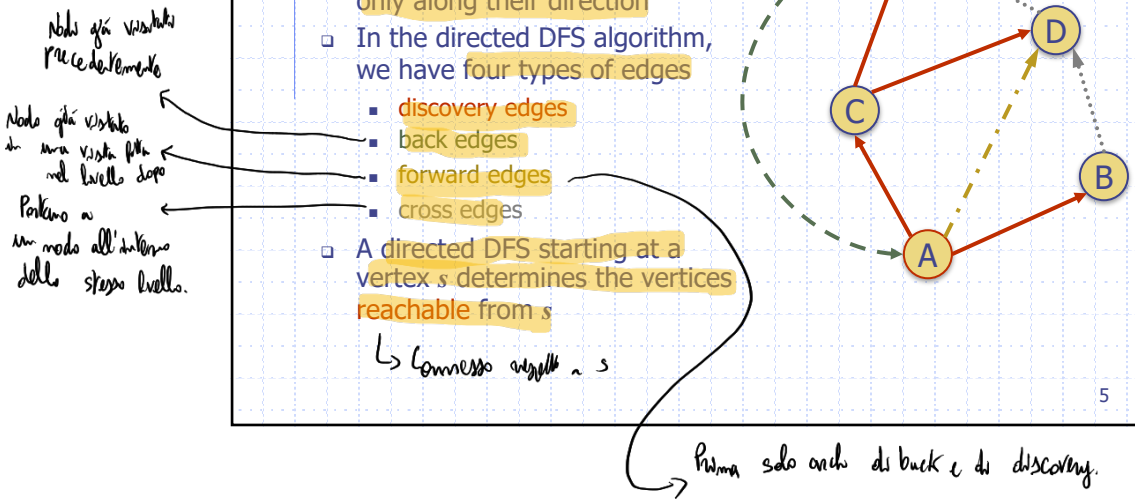
- Scheduling: edge (a,b) means task a must be completed before b can be started



4

Directed DFS

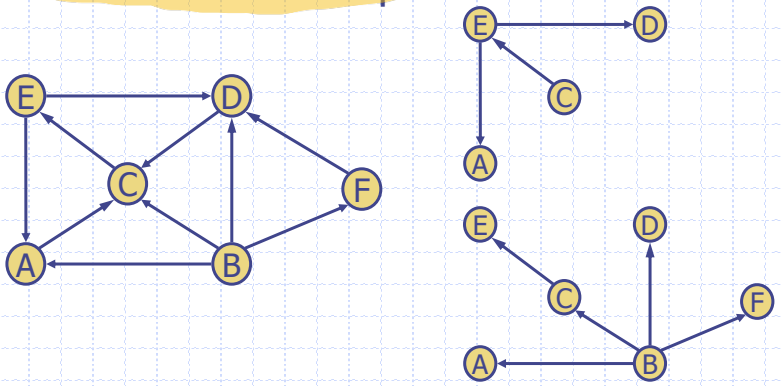
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex s determines the vertices reachable from s



5

Reachability

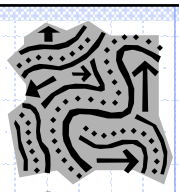
- DFS tree rooted at v: vertices reachable from v via directed paths



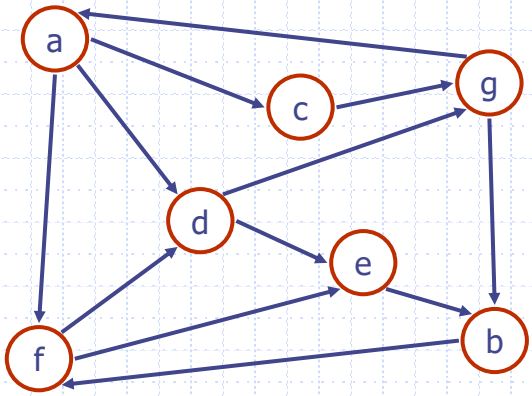
6

A partire da qualsiasi vertice posso raggiungere tutti gli altri vertici

Strong Connectivity



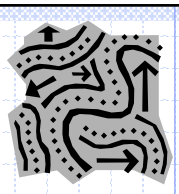
- Each vertex can reach all other vertices



7

Generalmente sono presi dei cicli a 3 vertici.

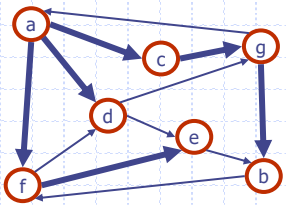
Strong Connectivity Algorithm



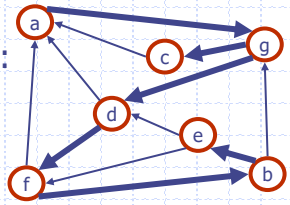
- Pick a vertex v in G
- Perform a DFS from v in G
  - If there's a w not visited, print "no"
- Let G' be G with edges reversed
- Perform a DFS from v in G'
  - If there's a w not visited, print "no"
  - Else, print "yes"
- Running time: O(n+m)

Puoi anche usare ingoing edges

G:

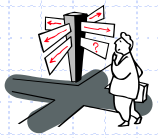


G' :

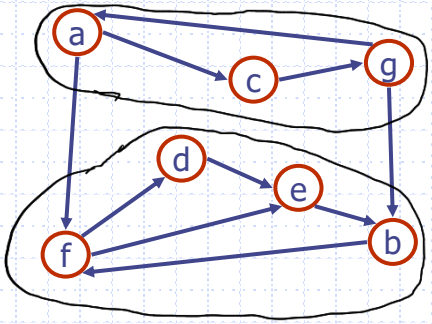


8

Strongly Connected Components



- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in O(n+m) time using DFS, but is more complicated (similar to biconnectivity).



{ a , c , g }

{ f , d , e , b }

9

non aggiungo f perché da f non posso raggiungere a, c, g.

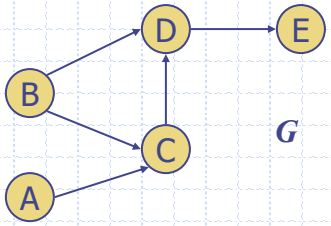
CHIUSURA TRANSITIVA

Transitive Closure

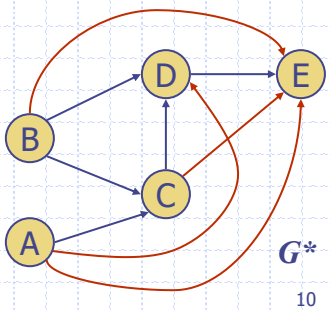
- Given a digraph G, the transitive closure of G is the digraph G^* such that
  - G^* has the same vertices as G
  - if G has a directed path from u to v ($u \neq v$), G^* has a directed edge from u to v

no self loop

- The transitive closure provides reachability information about a digraph



G



G^*

10

→ Almeno 1 vertice with no incoming edges e almeno 1 vertice with no outgoing edges

CICLO DIRETTO/ORIENTATO: HO 3 NODI CHE SI INCONTRANO L'UNO N'ELL'ALTRO

↳ NON HO CICLI IN GENERALE

DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

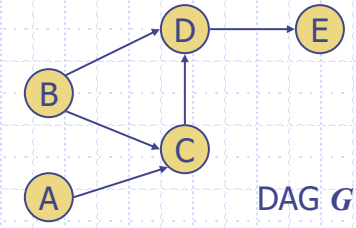
  $v_1, \ldots, v_n$

  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

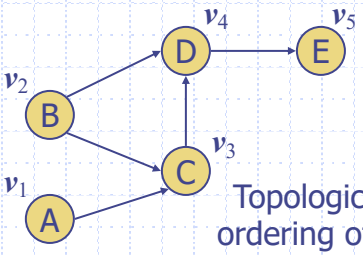
Theorem

A digraph admits a topological ordering if and only if it is a DAG

NON È UNICO!



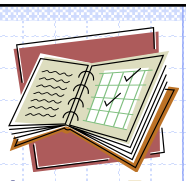
DAG G



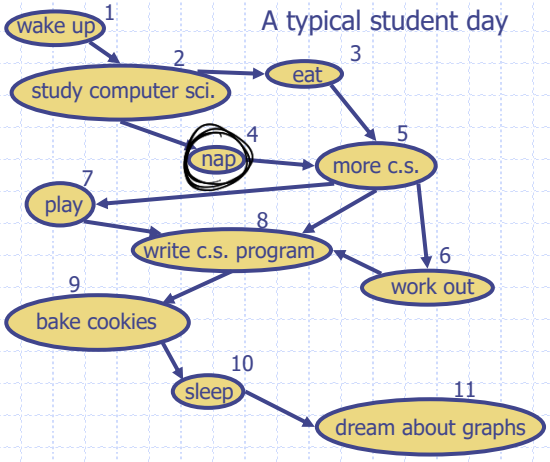
Topological ordering of G

11

Topological Sorting



- Number vertices, so that (u,v) in E implies u < v



12

Prima etichettavo vertici come visitati e anche di scoperta. Ora aggiungo etichetta numerica.

Implementation with DFS

- Simulate the algorithm by using depth-first search
- O(n+m) time.

```
Algorithm topologicalDFS(G)
    Input dag G
    Output topological ordering of G
  n ← G.numVertices()
  for all u ∈ G.vertices()
      setLabel(u, UNEXPLORED)
  for all v ∈ G.vertices()
      if getLabel(v) = UNEXPLORED
          topologicalDFS(G, v)
```

↳ Per tutti i vertici non esplorati parto.

```
Algorithm topologicalDFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the vertices of G
      in the connected component of v
  setLabel(v, VISITED)
  for all e ∈ G.outEdges(v)
      { outgoing edges }
    w ← opposite(v,e)
    if getLabel(w) = UNEXPLORED
        { e is a discovery edge }
      topologicalDFS(G, w)
    else
        { e is a forward or cross edge }
  Label v with topological number n
  n ← n - 1
```

↳ Differenza: etichetta

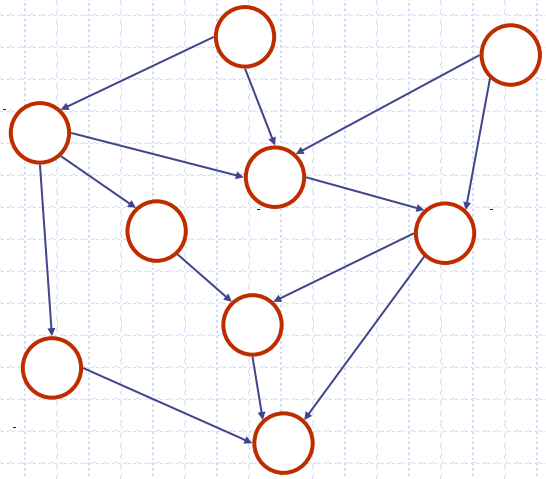
13

Inizialmente n è impostato al numero di vertici

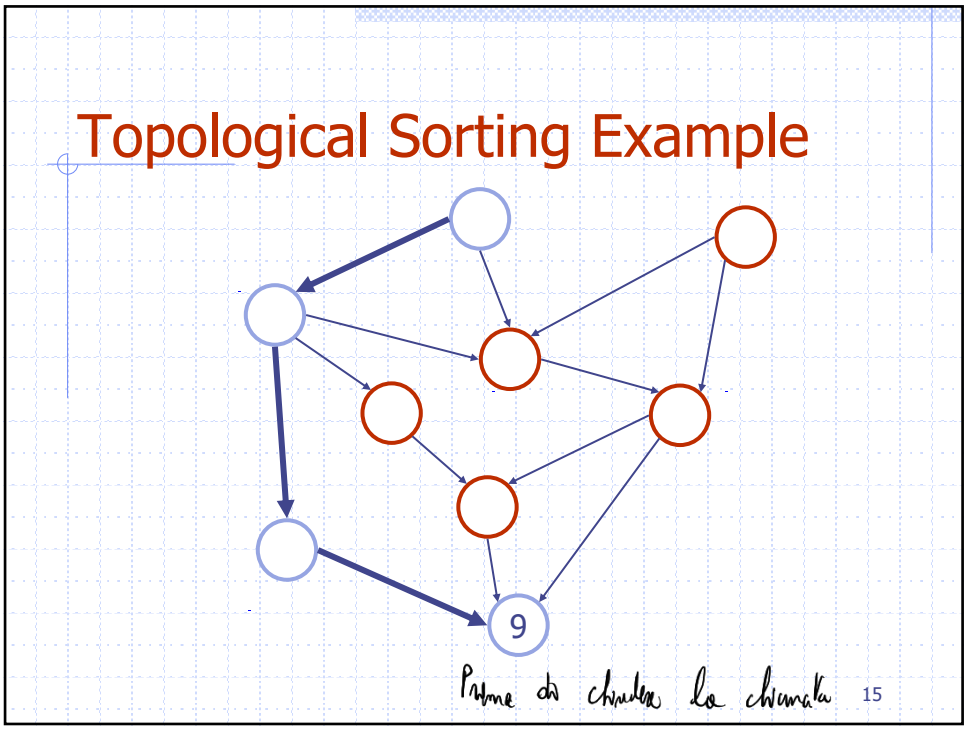
In teoria posso avere n diverse numerazioni. Ma si sceglie di partire da archi che non hanno ingoing edges perché altrimenti dobbiamo fare più iterazioni. Se ho grafo DAG esiste sempre modo che ha solo archi uscenti.

Con il numero topologico n. Quando con la DFS non posso più fare altro.

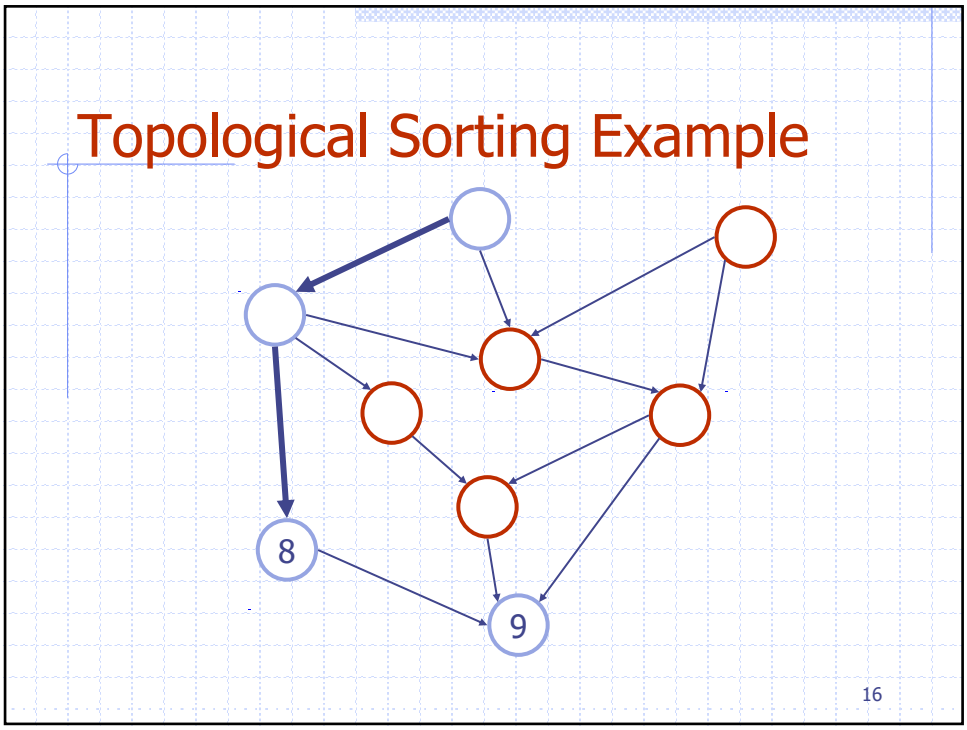
Topological Sorting Example

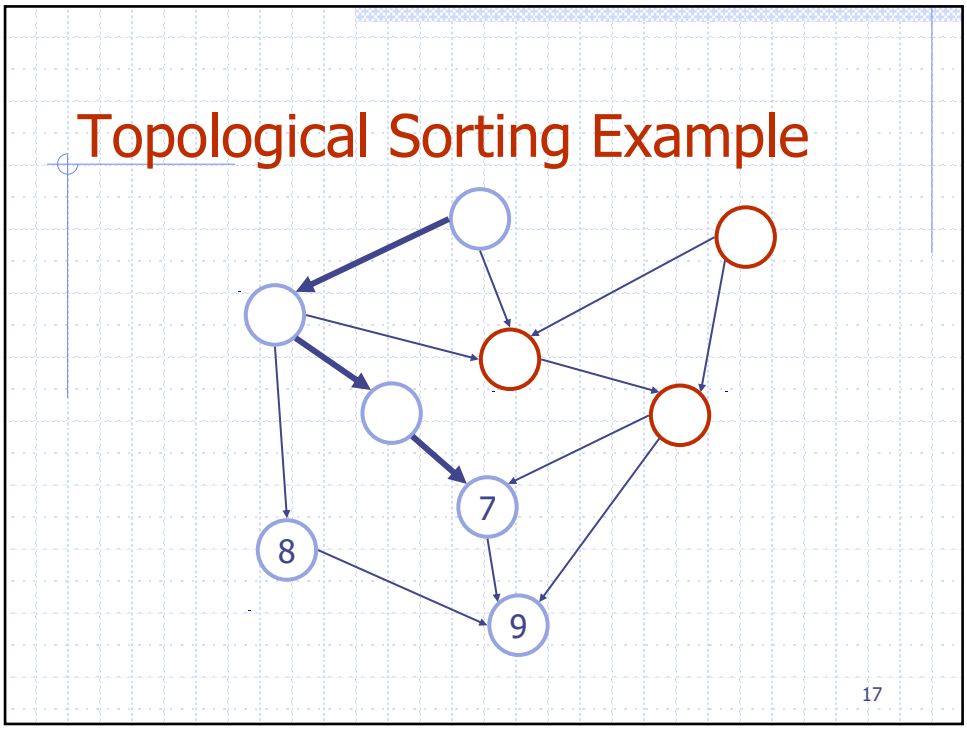


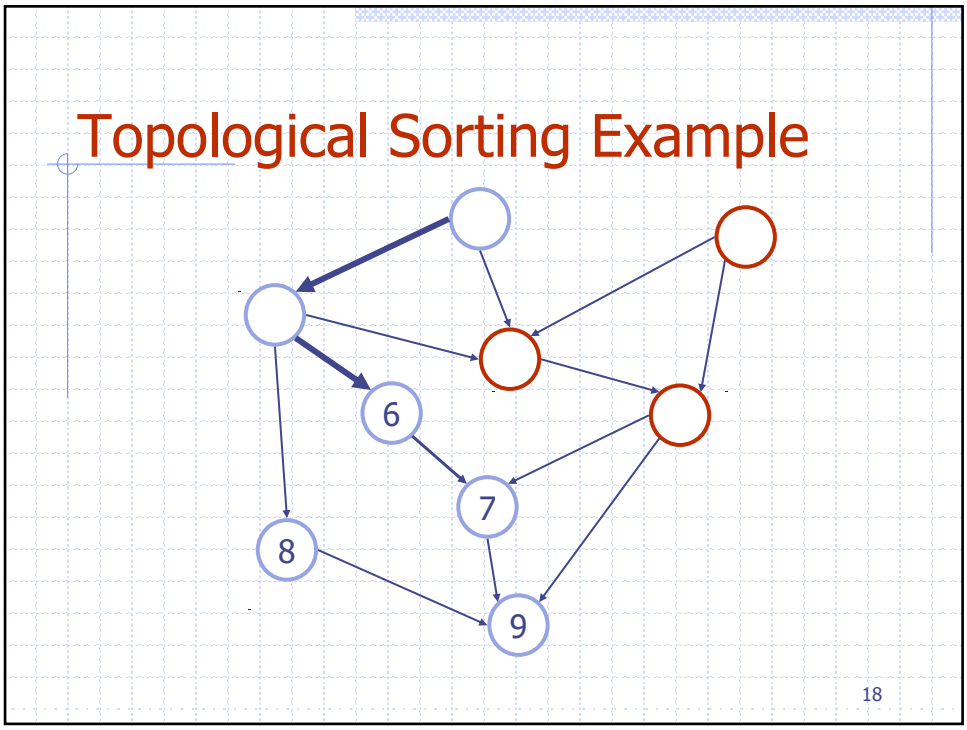
14

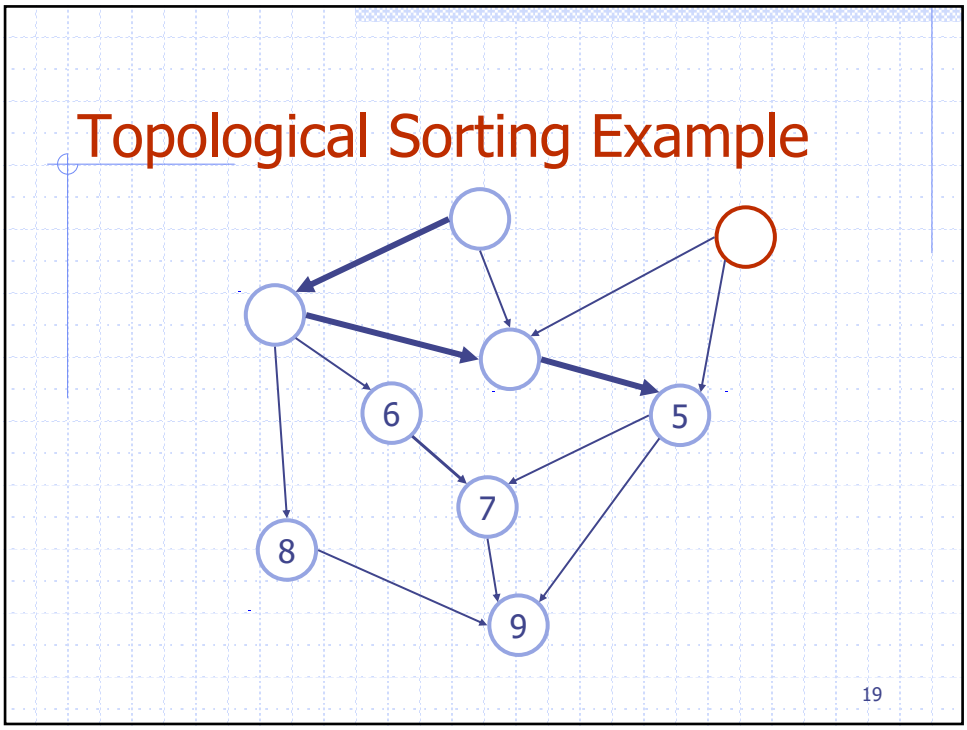


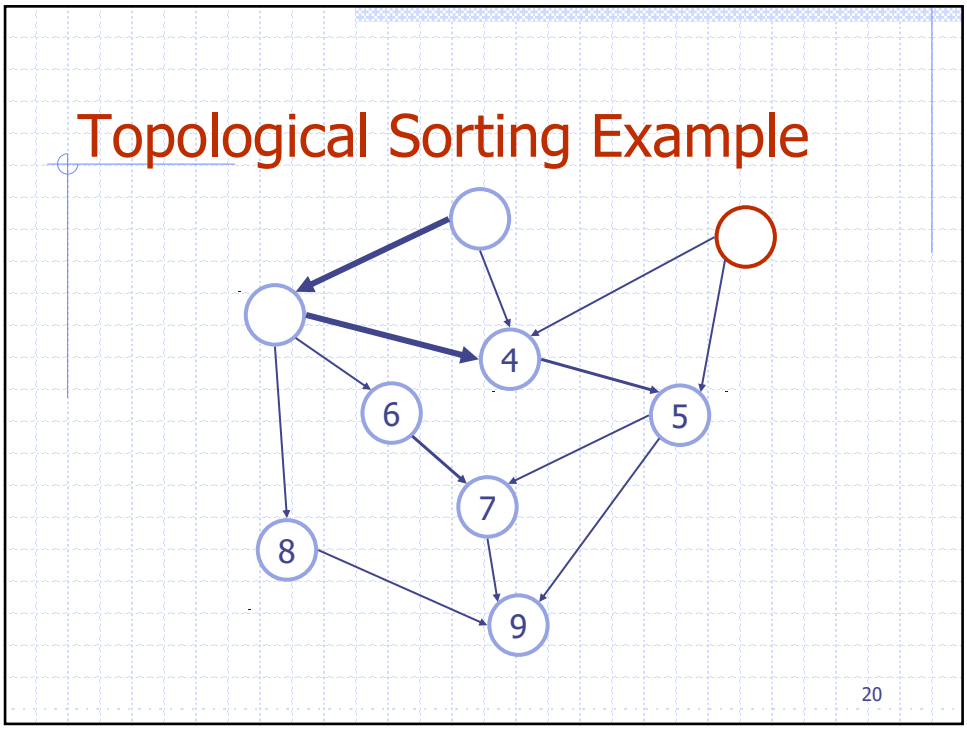
Prima di chiudere la chiamata marchia.

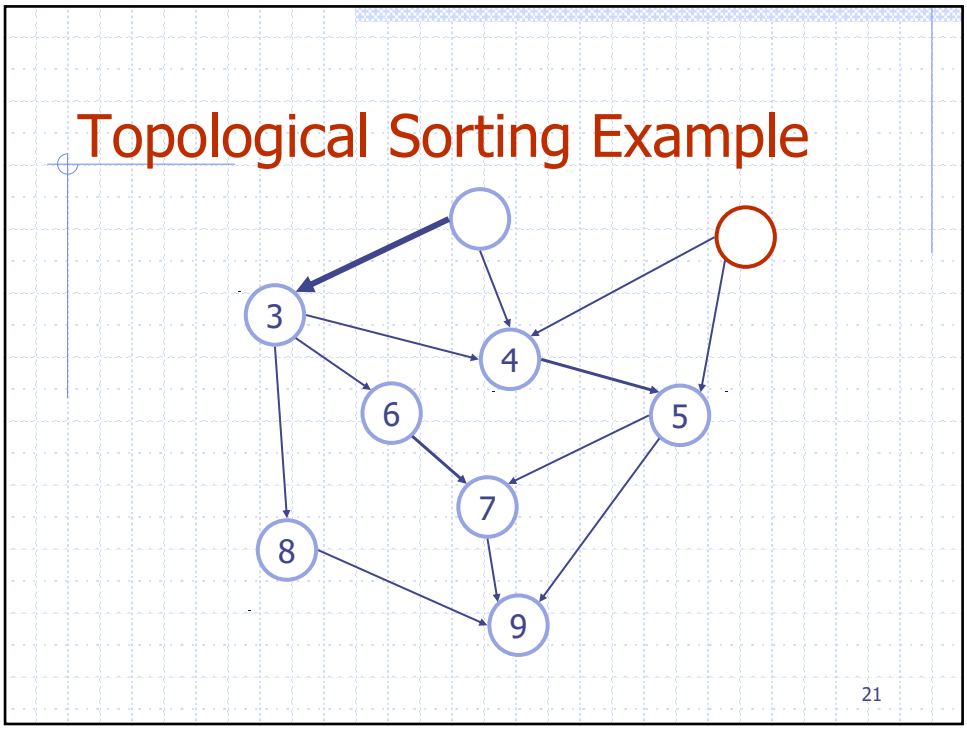


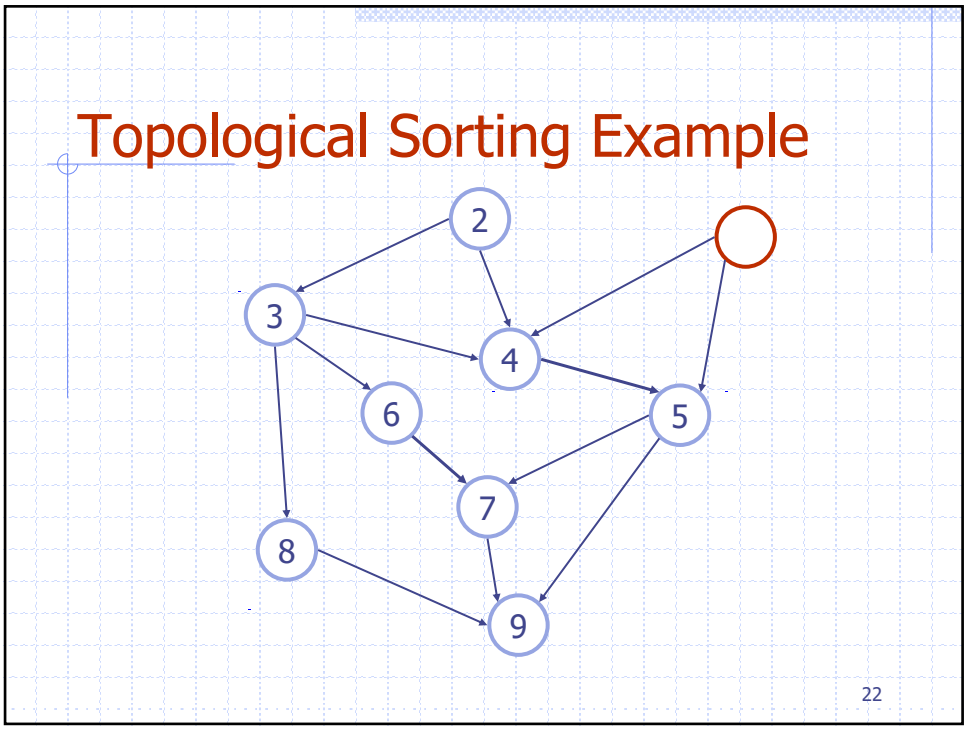

Topological Sorting Example
7
8
9
17


Topological Sorting Example
6
7
8
9
18

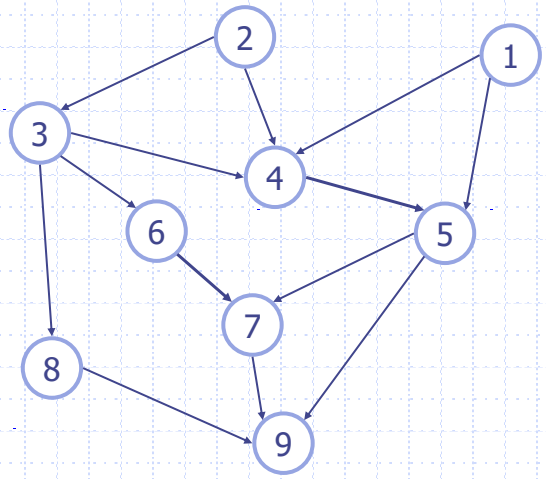

Topological Sorting Example
6
5
7
8
9
19


Topological Sorting Example
4
6
5
7
8
9
20


Topological Sorting Example
3
4
6
5
7
8
9
21


Topological Sorting Example
2
3
4
6
5
7
8
9
22

Topological Sorting Example

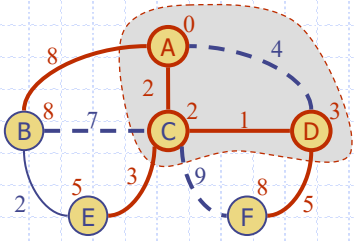


23

Se avessi scelto cose diverse su ③ avrei avuto cose pure diverse.
Non è fondamentale partire bene senza ingoing edges ma aiuta a livello di efficienza

Shortest Paths

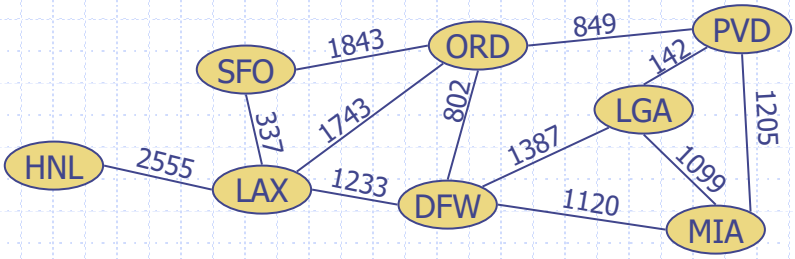
Non ho mai preso in considerazione nella BFS il peso degli archi per trovare percorso più breve.



24

Weighted Graphs

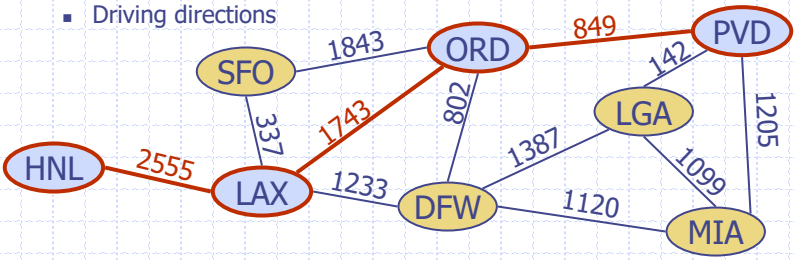
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



25

Shortest Paths

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



26

Somma dei pesi degli archi del percorso è la minima.

Shortest Path Properties

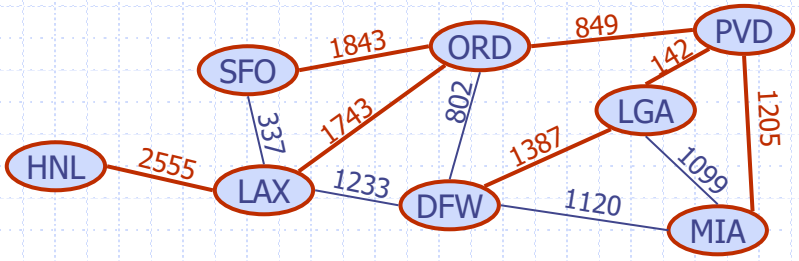
Property 1:
A subpath of a shortest path is itself a shortest path

Percorso intermedio

Property 2:
There is a tree of shortest paths from a start vertex to all the other vertices

↳ Albero di percorsi minimi a partire da quel vertice. → (Sottinteso che grafo sia connesso)

Example:
Tree of shortest paths from Providence



27

Dato vertice v, quali sono le distanze tra v e qualsiasi altro nodo?

Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

  ↳ Archi non in loop.
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

28

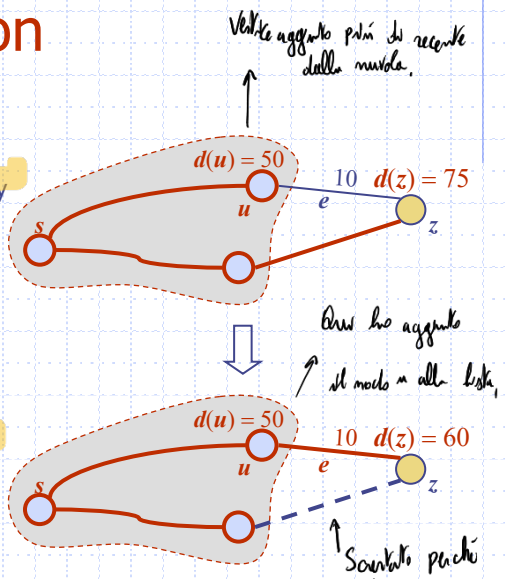
Non si può fare

Costruisco una mappa di vertici a partire dal vertice di partenza.

Edge Relaxation

- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

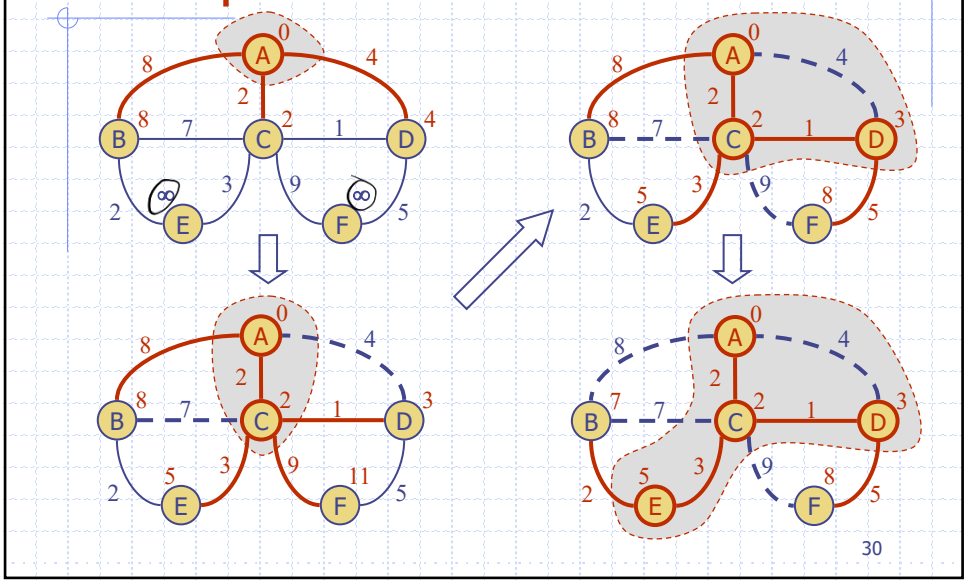
$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$



aggiorno etichetta su vertice z per la distanza.

1. Parto da A e creo nuvola. Distanza di A da sé stesso è 0.
2. Distanza da A ai vertici connessi sono il peso dell'arco.
3. La distanza dei vertici non connessi inizialmente ha peso infinito.

Example



Ora, peso più basso è CD. Includo D nella nuvola e aggiorno i pesi.

4. Passo da fare: dati archi uscenti da nuvola, scegli arco con peso minore.

Guardo arco che mi permette di raggiungere B, E, D, F.

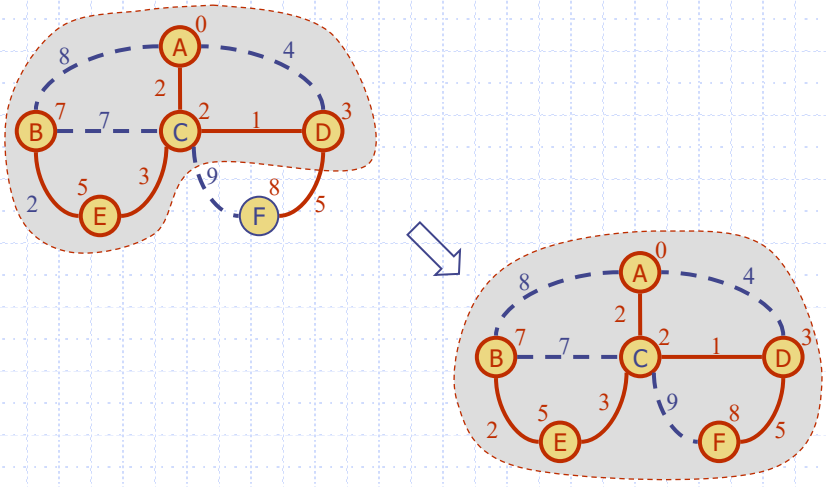
Aggiorno i pesi: non nuovo arco BC non mi permette di raggiungere B prima.

Invece CD mi fa arrivare prima. Arco AD bloccato.

Ora E ed F li posso a raggiungere

IN CONCLUSIONE:
PARTO DA NODO. SALTO SU NODO PIÚ VICINO.
AGGIORNO LE DISTANZE; SCARTO PERCORSI PIÚ LUNGHI SE CI SONO CONFLITTI
VADO SU NODO PIÚ VICINO. REPEAT

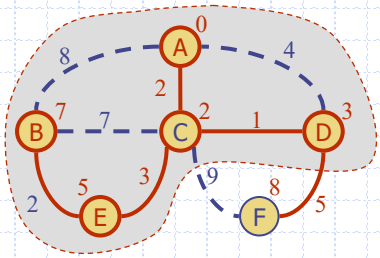
Example (cont.)



31

Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex

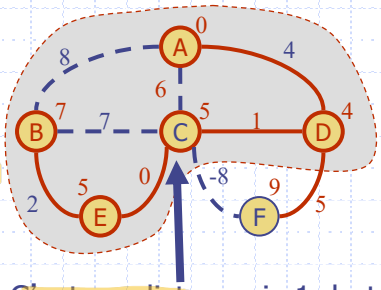


32

Se sbaglio, alla fine il risultato finale è sbagliato perché se ho percorso più breve, anche i sottopercorsi devono essere i più brevi possibili.
Non posso mai scegliere i più brevi sempre.

Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!

33

Ho già aggiunto C, che vale 5 come peso complessivo. Quando metto F, stampo FC con peso -8. E ora devo aggiornare pesi. Distanza CF è 1. Devo ricalcolare tutte le distanze, che vengono sottratte di -8. Ora devo riscegliere le distanze e ricontrollo. Provo a rivisitare -8.

Dijkstra's Algorithm

Algorithm ShortestPath(G, s): (e connesso)

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

Output: The length of a shortest path from s to v for each vertex v of G.

Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$. (Peso 0 a sorgente e poi altri a ∞)

Let a priority queue Q contain all the vertices of G using the D labels as keys.

while Q is not empty **do**

{pull a new vertex u into the cloud}

u = value returned by Q.remove_min() (etichetta minore: S)

for each vertex v adjacent to u such that v is in Q **do**

{perform the ***relaxation*** procedure on edge (u,v)}

if $D[u] + w(u,v) < D[v]$ **then**

$D[v] = D[u] + w(u,v)$

Change to $D[v]$ the key of vertex v in Q.

return the label $D[v]$ of each vertex v

→ Non dà il percorso, ma solo le distanze; grafi non sono statici, possono cambiare nel tempo

→ Posso ricostruire il percorso seguendo un ordine casuale e pesi fino a quando arrivo al nodo corretto.

→ Cos'è una coda a priorità? Coda normale, ma se ho 2 individui che entrano all'interno di una coda, c'è un'etichetta che può farmi entrare per primo in cima alla coda.
2 vertici con stessa etichetta ES. Ma vertice con peso più basso va avanti a tutto.

→ No priorità con D=∞ esclusa per D[s]=0. Nuvola inizialmente vuota

Analysis of Dijkstra's Algorithm

- Graph operations
  - We find all the incident edges once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list/map structure

35

STRUTTURE DATI DI SUPPORTO AGGIUNTIVE

Java Implementation

```
1  /** Computes shortest-path distances from src vertex to all reachable vertices of g. */
2  public static <V> Map<Vertex<V>, Integer>
3  shortestPathLengths(Graph<V,Integer> g, Vertex<V> src) {
4    // d.get(v) is upper bound on distance from src to v
5    Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
6    // map reachable v to its d value
7    Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
8    // pq will have vertices as elements, with d.get(v) as key
9    AdaptablePriorityQueue<Integer, Vertex<V>> pq;
10   pq = new HeapAdaptablePriorityQueue<>();
11   // maps from vertex to its pq locator
12   Map<Vertex<V>, Entry<Integer,Vertex<V>>> pqTokens;
13   pqTokens = new ProbeHashMap<>();
14
15   // for each vertex v of the graph, add an entry to the priority queue, with
16   // the source having distance 0 and all others having infinite distance
17   for (Vertex<V> v : g.vertices()) {
18     if (v == src)
19       d.put(v,0);
20     else
21       d.put(v, Integer.MAX_VALUE);
22     pqTokens.put(v, pq.insert(d.get(v), v));          // save entry for future updates
23   }
```

36

Java Implementation, 2

```
24   // now begin adding reachable vertices to the cloud
25   while (!pq.isEmpty()) {
26     Entry<Integer, Vertex<V>> entry = pq.removeMin( );
27     int key = entry.getKey( );
28     Vertex<V> u = entry.getValue( );
29     cloud.put(u, key);                          // this is actual distance to u
30     pqTokens.remove(u);                         // u is no longer in pq
31     for (Edge<Integer> e : g.outgoingEdges(u)) {
32       Vertex<V> v = g.opposite(u,e);
33       if (cloud.get(v) == null) {
34         // perform relaxation step on edge (u,v)
35         int wgt = e.getElement( );
36         if (d.get(u) + wgt < d.get(v)) {        // better path to v?
37           d.put(v, d.get(u) + wgt);             // update the distance
38           pq.replaceKey(pqTokens.get(v), d.get(v));  // update the pq entry
39         }
40       }
41     }
42   }
43   return cloud;          // this only includes reachable vertices
44 }
```

37

Bellman-Ford Algorithm (not in book)

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists

Algorithm *BellmanFord*(*G*, *s*)
  for all $v \in G.vertices()$
    if $v = s$
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    for each $e \in G.edges()$
      { relax edge e }
      $u \leftarrow G.origin(e)$
      $z \leftarrow G.opposite(u,e)$
      $r \leftarrow getDistance(u) + weight(e)$
      if $r < getDistance(z)$
        setDistance(z,r)

INIZIALIZZAZIONE

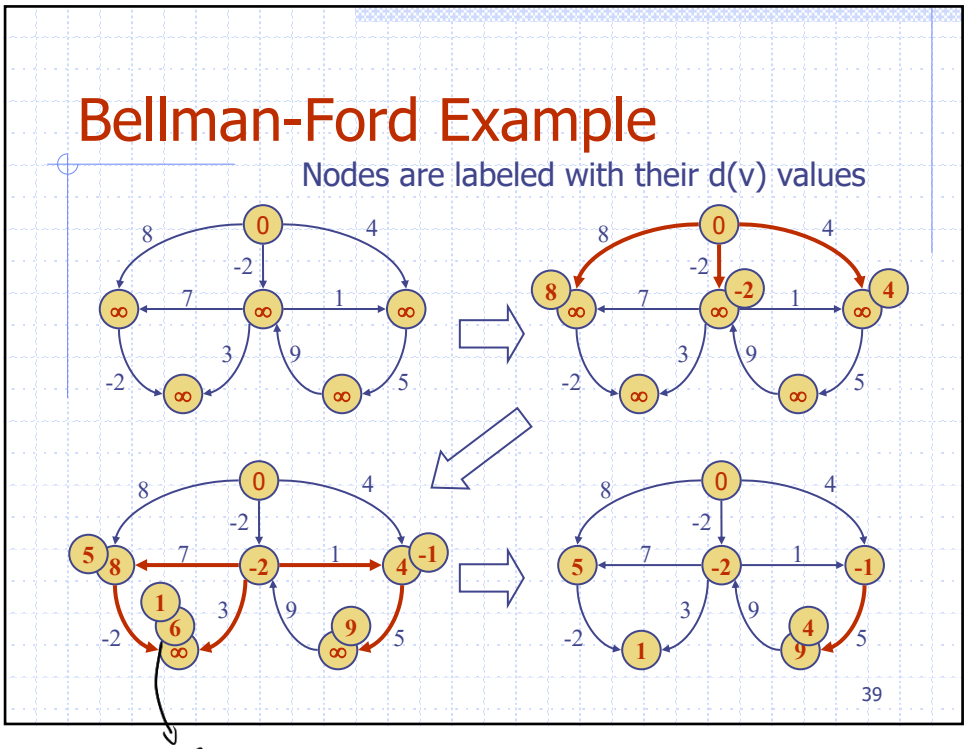
PRIMA DICEVAMO → PER TUTTI GLI ARCHI DELLA RETE

RILASSA ARCO PER TUTTI GLI ARCHI

A ogni passo rilasso tutti gli archi. Potrei non fermarmi mai? sì. Mi fermo a n-1 step.

N rilassamenti per n vertici

38



Non proseguo con gli altri perché avevo pesi infiniti ⇒ ∞+3 non posso confrontarlo con ∞.

Parto prima con 8 -> -2 -> 6. Poi cambia.

Ora posso confrontare 8-2=6 con peso ∞. Più piccolo.

Mi fermo dopo 5 iterazioni.

(Parto sempre da peso più piccolo?)