La Sapienza
Università degli Studi di Roma
Dipartimento di Informatica e Sistemistica

Computer Networks II
Graph theory and routing algorithms

Luca Becchetti
Luca.Becchetti@dis.uniroma1.it
A.A. 2009/2010

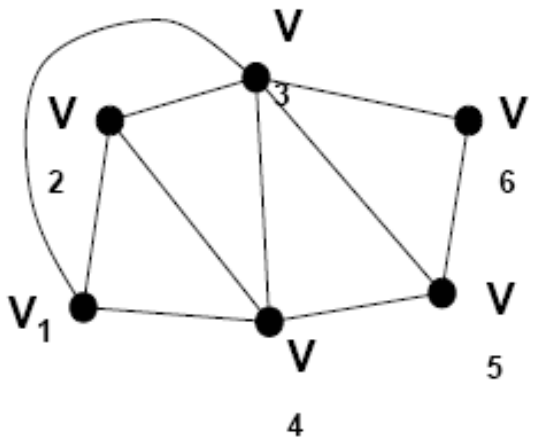
Graphs

- G(V, E) where:
  - V is vertex set
  - E is edge set: every e in E connects two vertices
  - If (i, j) connected they are said *adjacent*
  - Edge e = (i, j) is said *incident* to i and j
  - |V| = cardinality of the vertex set

Graphs/cont.

- A graph G(V, E) can be represented by |V| x |V| *adjacency matrix*

$$A = [a_{ij}] \qquad a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$



	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	1	1	1	0	0
V_2	1	0	1	1	0	0
V_3	1	1	0	1	1	1
V_4	1	1	1	0	1	0
V_5	0	0	1	1	0	1
V_6	0	0	1	0	1	0

Graphs/cont.

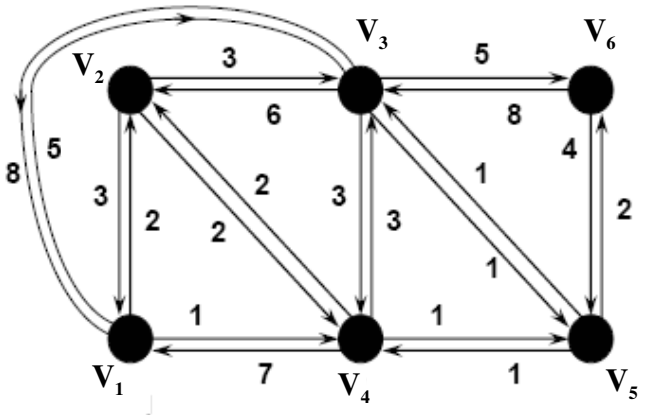
- A **path** between vertices i and j is a sequence of vertices and edges starting with vertex i and ending in j, such that every edge is incident to the preceding and following vertices
- A **path** is **simple** if every node/edge appears only once
- **Distance** between vertices i and j: minimum number of edges along a path from i to j
- **Cycle**: simple path where starting vertex coincides with ending vertex
- A graph is **connected** if there exists a path between any possible vertex pair i and j
- A graph is **directed** if edges have a direction. In this case, (i, j) belonging to E *does not* imply that (j, i) belongs to E. Edges are called *arcs* in this case
- A graph is **weighted** if every edge (or arc) comes with a number (its **weight** or **metric**)

Graphs/cont.

- Adjacency matrix of a directed (weighted) graph:

$$A = [a_{ij}] \qquad a_{ij} = \begin{cases} W_{ij} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- The **length** of a path in a weighted graph is the sum of the weights of its edges (arcs)
- Example:



Directed weighted graph

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	2	5	1	0	0
V_2	3	0	3	2	0	0
V_3	8	6	0	3	1	5
V_4	7	2	3	0	1	0
V_5	0	0	1	1	0	2
V_6	0	0	8	0	4	0

Graph theory/cont.

- Any packet network can modelled as a directed weighted graph:
  - Nodes are the routers
  - Arcs are subnets
- Routing function for a packet equivalent to finding shortest path in the graph associated to the network
  - Minimum number of hops (unweighted graph)
  - Shortest path (weighted graph)

Trees

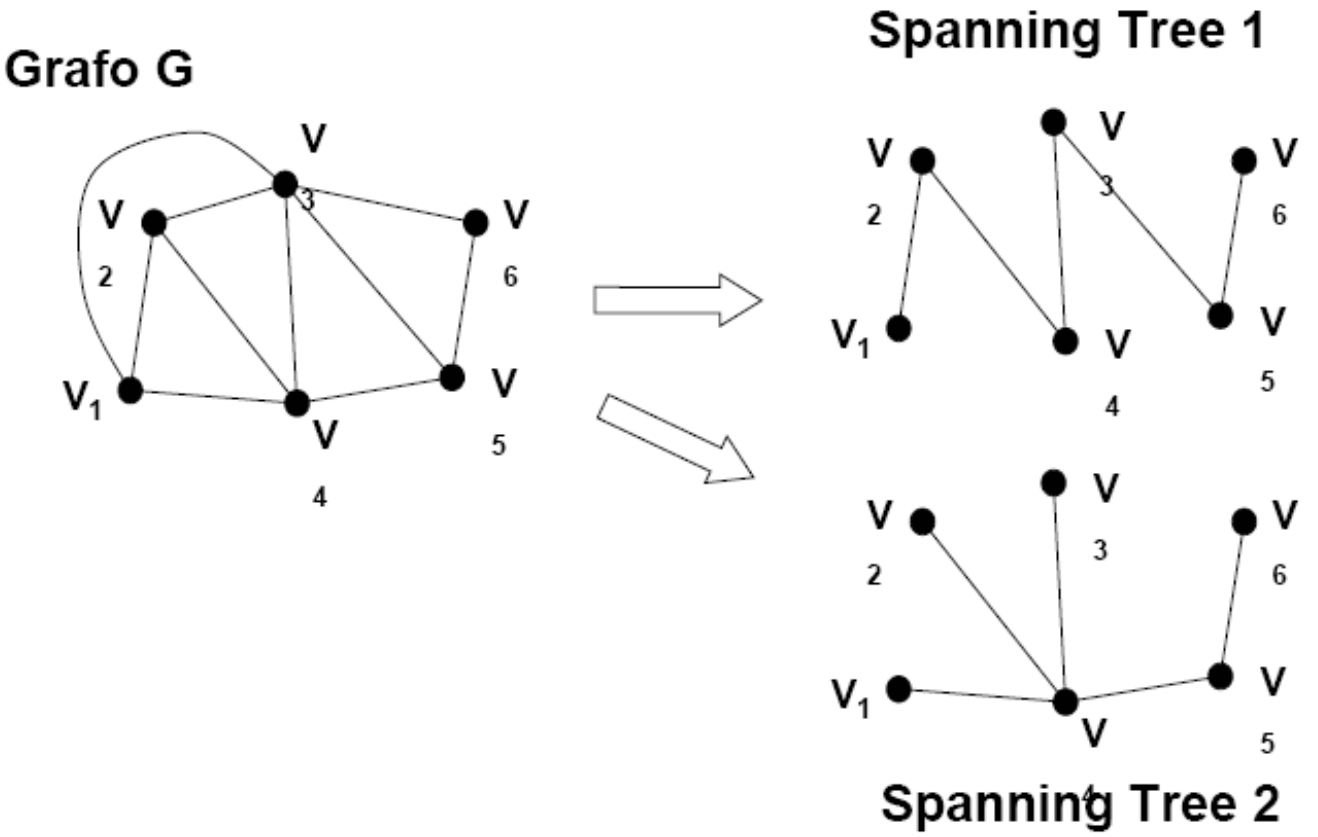
- Graph T is a tree if:
  - One and only one simple path between every vertex pair (i, j)
  - If N = |V|:
    - Exactly N-1 edges (arcs)
    - Connected, no cycles
- Every vertex of a tree can be assumed to be the **root** of the tree
- A tree can be represented by arranging vertices in sequential layers of increasing distance from root
- In a layered representation of a tree
  - Every vertex (except for the root), has only one **parent vertex**
  - Every vertex has 0 or more **children**
  - A node with 0 children is a **leaf**

Spanning trees

- A **subgraph** of a graph G(V,E) is obtained as follows:
  - G'(V', E'), where V' and E' are subsets of V and E respectively
  - For every arc (edge) e belonging to E', both i and j must belong to V'
- A subgraph T(V', E') of G(V, E) is a **spanning tree** for G if:
  - T is a tree
  - V' = V
- There is no unique spanning tree in general

Spanning trees

- Example:

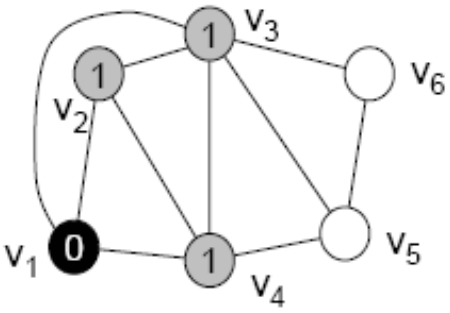


Spanning trees

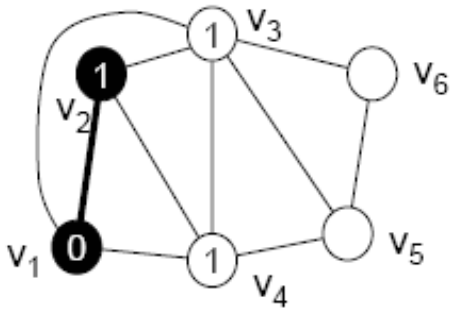
- Searching a spanning tree: **Breadth-First Search (BFS) algorithm**
  - Basic idea: explore vertices levelwise starting from root
  - Algorithm:
    - Define a root vertex (let it be x)
    - Explore vertices adjacent to x (level 1 vertices)
    - For every vertex belonging to level 1: explore adjacent vertices not explored during previous step (level 2 vertices)
    - Iterate until all vertices are reached

Breadth-First Search algorithm

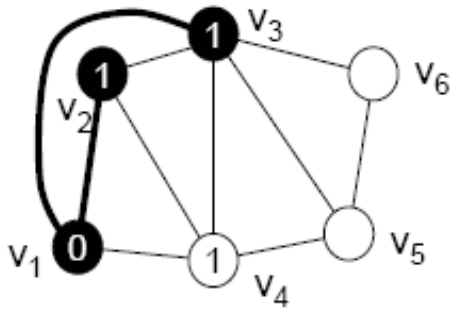
- Example:



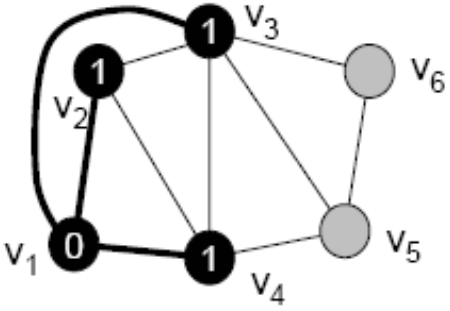
T={1}



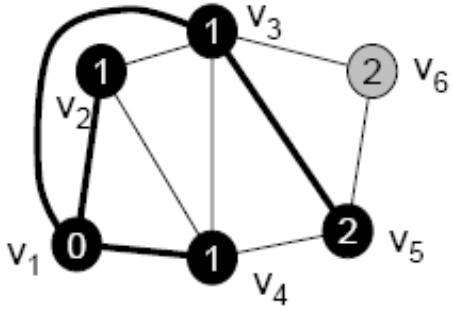
T={1,2}



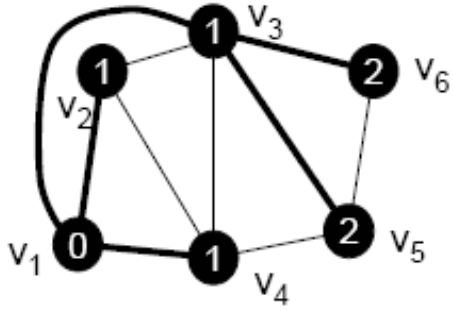
T={1,2,3}



T={1,2,3,4}



T={1,2,3,4,5}



T={1,2,3,4,5,6}

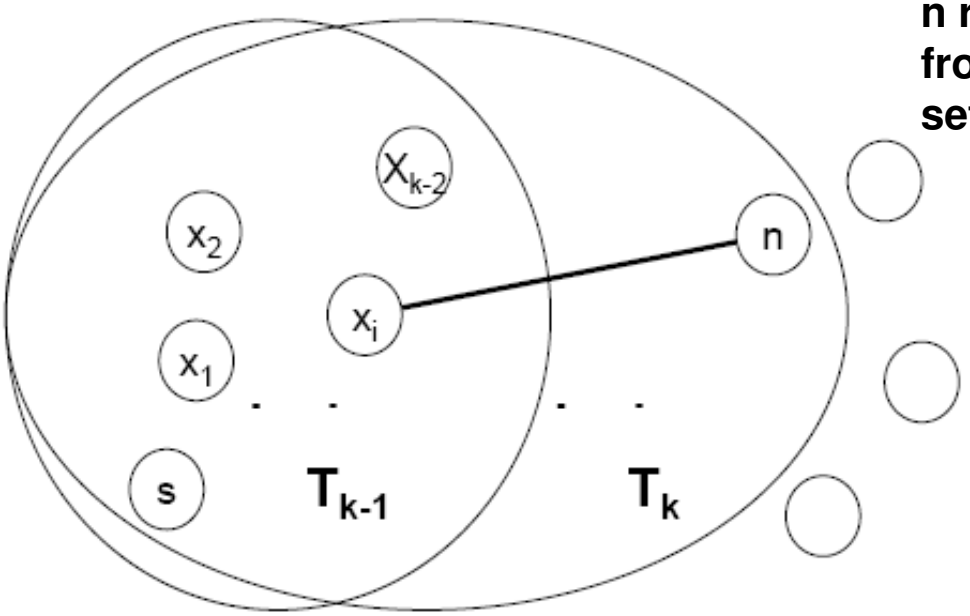
Breadth-First Search algorithm

- In an *unweighted* graph the BFS algorithm:
  - Produces a spanning tree
  - Finds shortest paths from root to all possible destinations
    - All vertices at level 1 have minimum distance from root; those at level 2 have distance 2 (otherwise they would be of level 1) ...
- Computational complexity of BFS
- $O(|V| \cdot |E|)$
  - Worst case (complete graph):
    - If $|V|=n \rightarrow |E| = n(n-1) \rightarrow$ complexity is $O(|V|^3)$

Dijkstra's algorithm

- In a *weighted* graph: find shortest path between source vertex s and all other vertices in the graph
- Algorithm proceeds in sequence of consecutive steps:
  - At step k: found k nodes reachable from s at minimum cost; denote by T_k this set
  - At step k+1: find vertex v at minimum distance from s among those that are reachable over paths that only traverse vertices in T_k (with the exception of v itself)
  - Set $T_{k+1} = T_k \cup \{v\}$
  - Algorithm terminates when all nodes explored (i.e., they have a finite distance)

Dijkstra's algorithm



Step k: $T_k = T_{k-1}$ {n}, where:

n reachable with shortest path from source using *only* nodes in set T_{k-1} (except n)

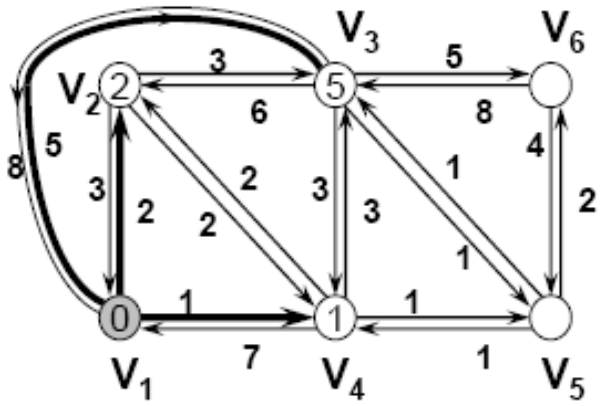
Situation at step k

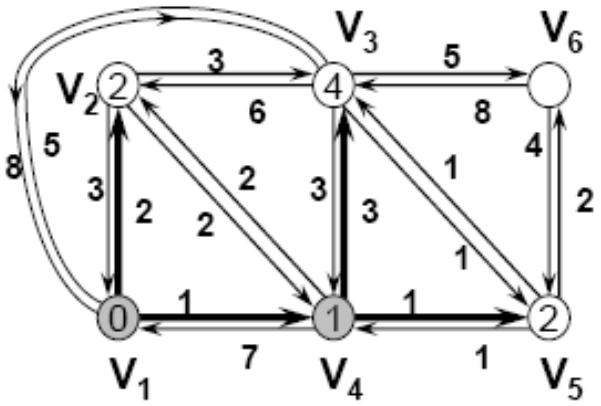
Dijkstra's algorithm

- Notation:
  - V : node/vertex set
  - N=|V|
  - s : source node
  - T_k : set of nodes reached at the end of step k of the algorithm
  - w(i,j) : weight (cost) of link (arc, edge) (i,j)
    - w(i,i) = 0
    - w(i,j) ≥ 0 if vertices i and j are adjacent
    - w(i,j) = ∞ if vertices i and j are adjacent
  - $L_k(n)$: cost of shortest path from s to generic node n, computed by algorithm until step k of the algorithm
    - Defined also for nodes not belonging to T_k

Dijkstra's algorithm - Example

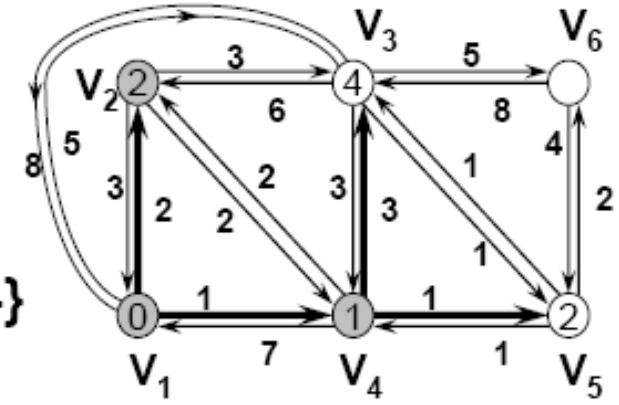
- Example 1/2:





$T_1 = \{1\}$

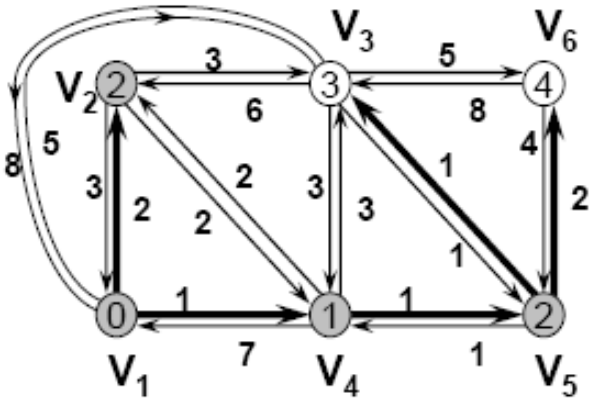
$T_2 = \{1,4\}$

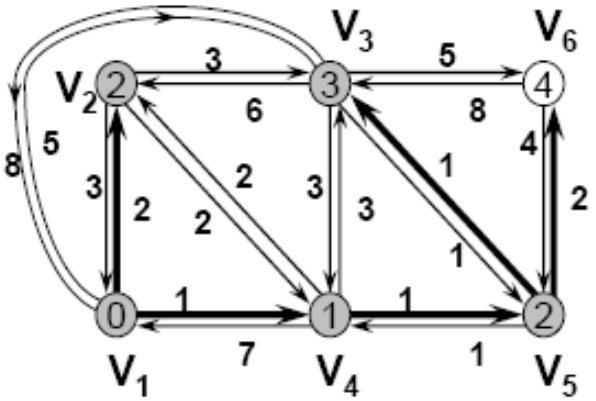


$T_3 = \{1,2,4\}$

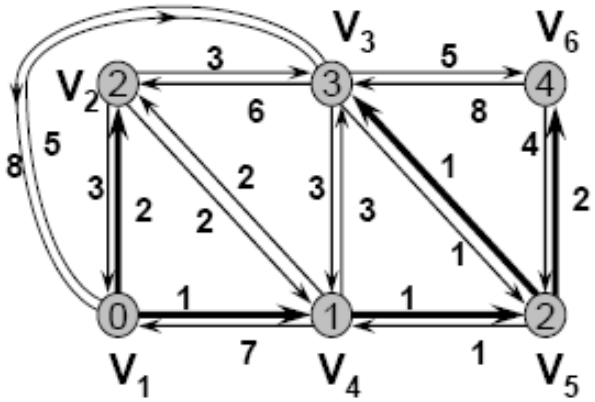
Dijkstra's algorithm - Example

- Example 2/2:





$T_4 = \{1,2,4,5\}$



$T_5 = \{1,2,3,4,5\}$

$T_6 = \{1,2,3,4,5,6\}$

Dijkstra's algorithm

- Initialization (k=1)
  - $T_1 = \{s\}$
  - $L_1(n) = w(s,n)$ for $n \neq s$
- Adding a new node (step $1 \leq k \leq N$)
  - Find $x \notin T_{k-1}$ such that:

$$L_{k-1}(x) = \min_{j \notin T_{k-1}} L_{k-1}(j)$$

  - Add x to T_{k-1}
  - Remark: x added is either directly connected to s, or it is connected to a node in T_{k-1}; this follows becaues only in such 2 cases $L_{k-1}(x) \neq \infty$ holds
- Update current shortest paths:
  - $L_k(n) = \min [L_{k-1}(n), L_{k-1}(x) + w(x,n)]$ fo all $\mathbf{n \notin T_k}$

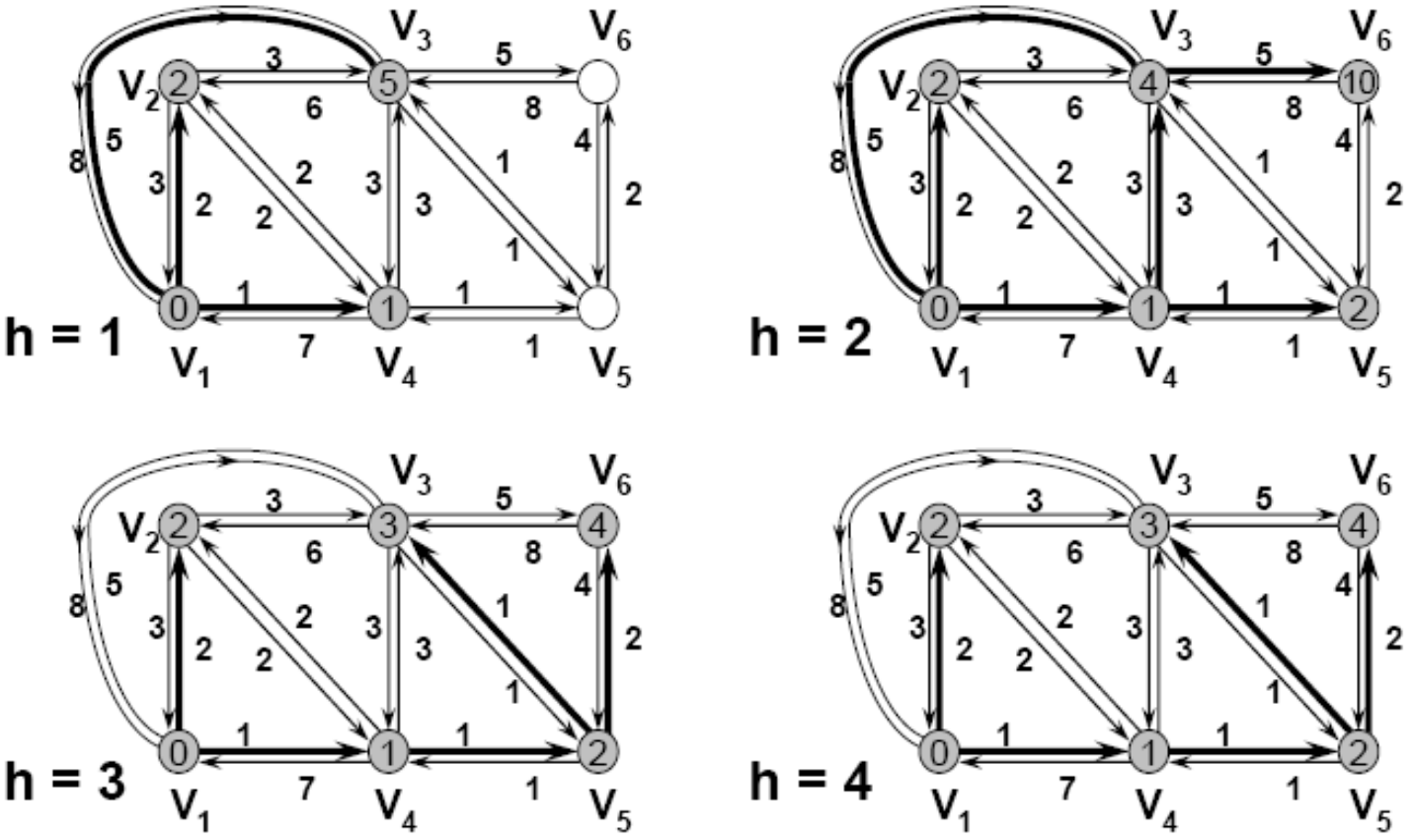
Dijkstra's Algorithm

- At termination:
  - Set T_N is a spanning tree of the original graph. It contains shortest paths between s and all other vertices in the graph
  - $L_N(n)$ ∀ n in V is the shortest path cost between s and generic node n
- Note that:
  - At step k, k-th node added to T_{k-1} and shortest path between s and such node is found
  - If node added at k-th step is n, this shortest path only uses nodes belonging to T_{k-1} (with the exception of n)
- Algorithm's complexity is $O(|V|^2)$; more efficient implementations can achieve $O(|V| \cdot \log|V|)$

Bellman-Ford's algorithm

- Like Dijkstra's: for a source node s, finds shortest path from s to all other vertices in the graph G
- Sequence of steps:
  - First step: find shortest paths between s and other vertices, **such that these paths are at most 1 hop long**
  - Second step: find shortest paths between s and other vertices, **such that these paths are at most 2 hops long**
  - Iterate until shortest paths have a number of hops that is at most the **diameter of the graph**
    - **Diameter**: maximum distance between any pair of vertices in the graph (measured in **hops**)

Bellman-Ford's algorith - Example



Bellman-Ford's algorith

- Notation:
  - V : node/vertex set
  - N=|V|
  - s : source node
  - T_k : set of nodes reached at the end of step k of the algorithm
  - w(i,j) : weight (cost) of link (arc, edge) (i,j)
    - w(i,i) = 0
    - w(i,j) ≥ 0 if vertices i and j are adjacent
    - w(i,j) = ∞ if vertices i and j are adjacent
  - $L_h(s,n)$: cost of shortest path between s and n found until step h, with the following constraint: the shortest path is at most h hops long

Algoritmo di Bellman-Ford

- initialization:
  - $L_0(n) = \infty$ ▪ $n \neq s$
  - $L_h(s) = 0$ h
- Update:
  - For every $h \geq 0$ for every node n:

$$L_{h+1}(s,n) = \min_j \left[L_h(s,j) + w(j,n) \right]$$

  - Connect n to predecessor node j that achieves minimum. Disconnect n from other predecessor nodes found in previous iterations
- Complexity is $O(|V|.|E|)$, so in the worst case, when $|E|=|V|^2$, $O(|V|^3)$

Bellman-Ford's algorithm

- Bellman-Ford's solution is described by the following recursion:

$$L(s,n) = \min_{j \in A_n} \left[L(s,j) + w(j,n) \right]$$

A_n: set of vertices that are predecessors of n in G

  - Shortest path between s and n is given by concatenation of shortest path between s and one predecessor j of n and the link between j and n

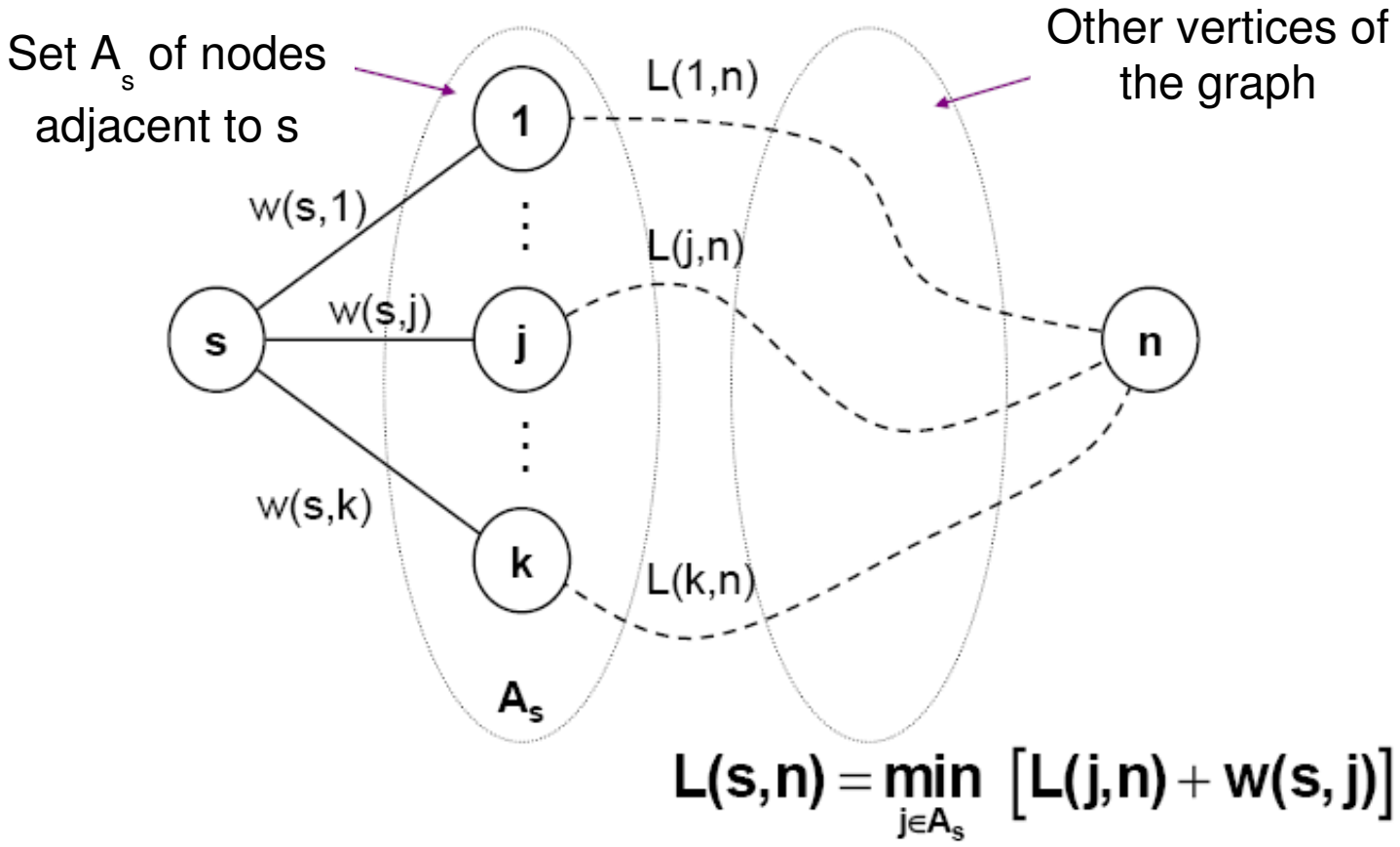
- Also:

$$L(s,n) = \min_{j \in A_s} \left[w(s,j) + L(j,n) \right]$$

A_s: set of vertices that are adjacent to s in G

  - **Shortest path between s and n is given by minimum cost concatenation of a shortest path between a node j adjacent to s and n and the link between s and j**

Bellman-Ford's algorithm



$$L(s,n) = \min_{j \in A_s} \left[L(j,n) + w(s,j) \right]$$

Dijkstra vs. Bellman-Ford

- Both algorithms compute shortest path tree
- Bellman-Ford's algorithm has higher worst-case complexity; algorithms are equivalent in practice
- Dijkstra's algorithm requires that source node **knows topology: all arcs/edges and all weights**
  - Need to exchange information with all other nodes
- Bellman-Ford's algorithm requires **knowledge of weights (state) of arcs to adjacent vertices and costs of shortest paths starting at adjacent vertices**
  - Possible to have communication only between adjacent vertices (distributed implementation)

Bellman-Ford's algorithm – Distributed implementation

- Every node s **asynchronously** executes following computation:

$$L(s,n) = \min_{j \in A_s} \left[L(j,n) + w(s,j) \right]$$

  - Using:
    - Values w(s,j) (j in A_s) which s **directly knows**
    - Values L(j,n) (j in A_s) **received from adjacent nodes** [s uses most recently received estimations]
- s sends updates to its neighbours whenever its estimated shortest path distance to another vertex in the network changes
- Distributed implementation converges to solution if changes in the network occur more slowly than convergence speed of algorithm
  - **"Bad news phenomenon"** problems may occur
    - Convergence can be very slow in some cases
    - Number of iterations can be high

References

- J. F. Kurose and K. W. Ross. Computer Networking: A Top-Down Approach, 4/E, Chapter 4
- T. Cormen et al. Introduction to Algorithms, 3/E, Chapter 25
- Also can find good treatment of shortest path algorithms starting here: http://en.wikipedia.org/wiki/Routing