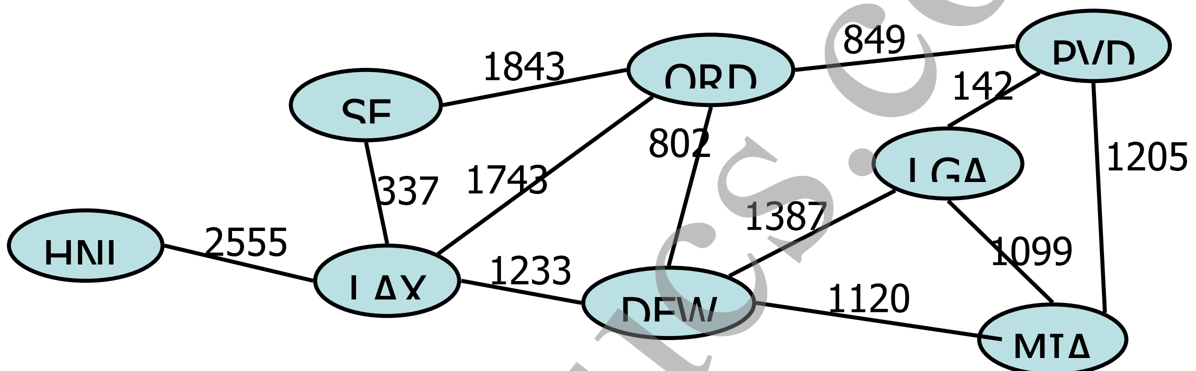
Graph Theory and Combinatorics

Weighted Graphs

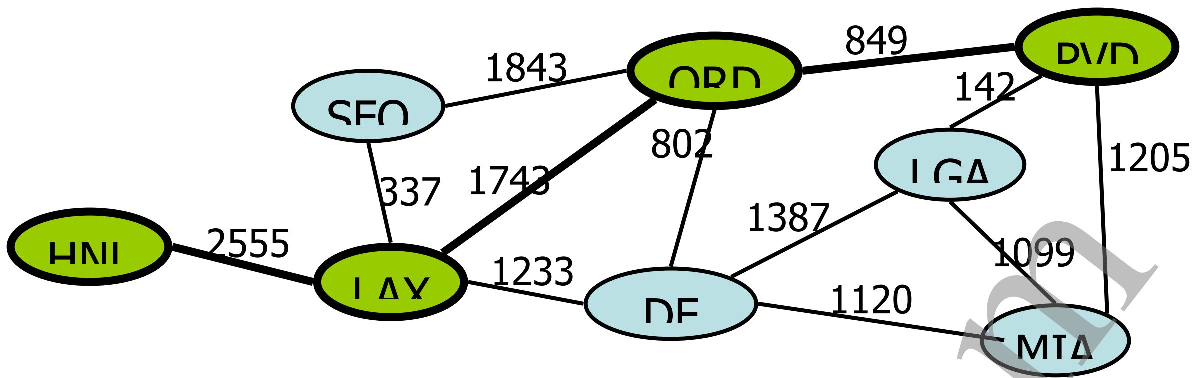
- **In a weighted graph, each edge has an associated numerical value, called the weight of the edge**
- **Edge weights may represent, distances, costs, etc.**
- **Example:**
  - **In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports**



Shortest Path Problem

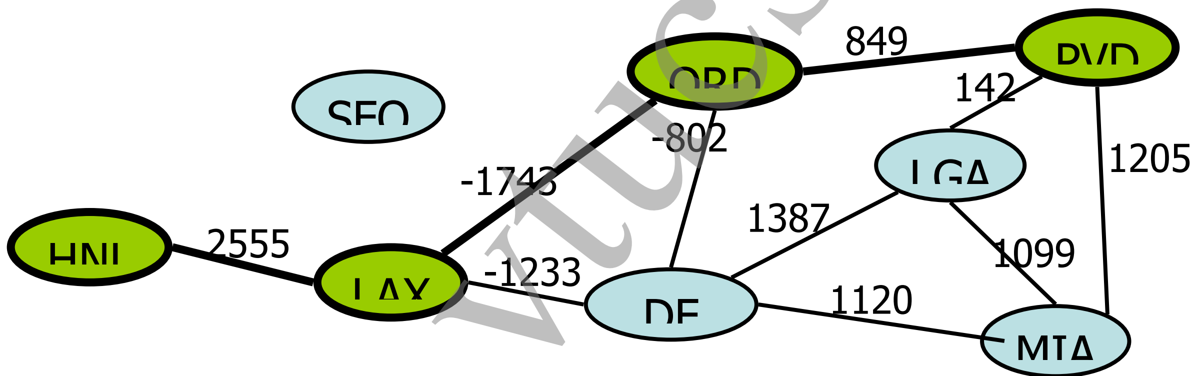
- **Given a weighted graph and two vertices *u* and *v*, we want to find a path of minimum total weight between *u* and *v*.**
  - **Length of a path is the sum of the weights of its edges.**
- **Example:**
  - **Shortest path between Providence and Honolulu**
- **Applications**
  - **Internet packet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

www.vtucs.com



Shortest Path Problem

- If there is no path from *v* to *u*, we denote the distance between them by *d*(*v*, *u*)=+▪
- What if there is a negative-weight cycle in the graph?



Shortest Path Properties

Property 1:

A subpath of a shortest path is itself a shortest path

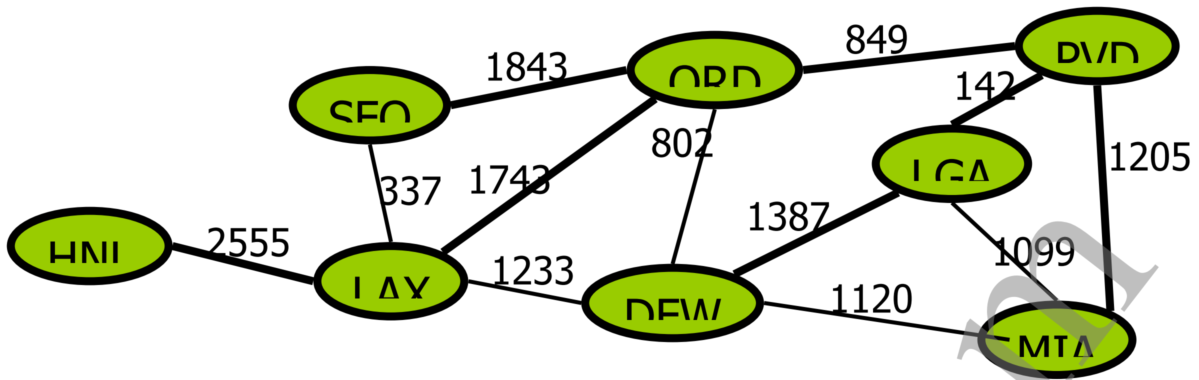
Property 2:

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

Example:

Tree of shortest paths from Providence

www.vtucs.com



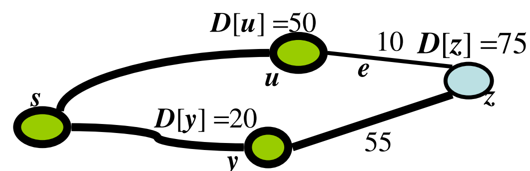
Dijkstra's Algorithm

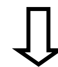
- The **distance** of a vertex ***v*** from a vertex ***s*** is the length of a shortest path between ***s*** and ***v***
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex ***s*** (**single-source** shortest paths)
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- 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
- The label ***D*[*v*]** is initialized to positive infinity
- At each step
  - We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label, ***D*[*v*]**
  - We update the labels of the vertices adjacent to ***u*** (i.e. edge relaxation)

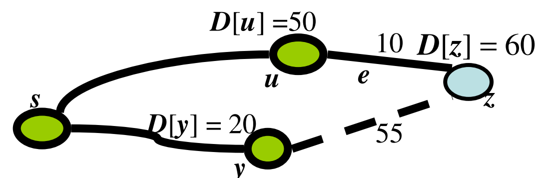
Edge Relaxation

- 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*]←min{***D***[*z*],***D***[***u***]+***weight***(***e***)}







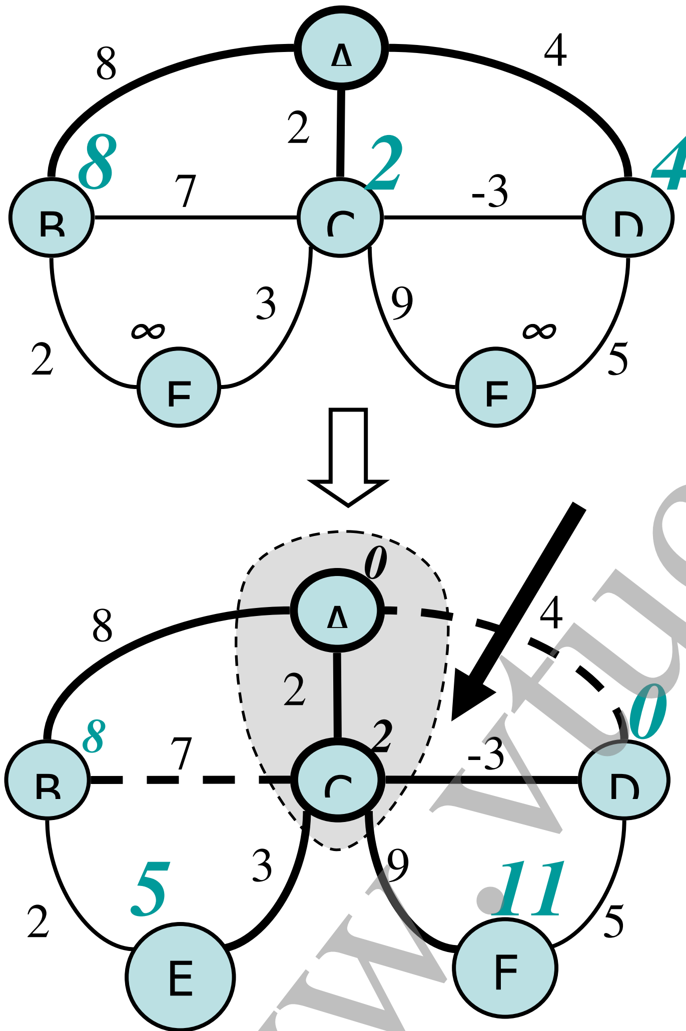
www.vtucs.com

Example

www.vtues.com

Doesn't Work for Negative- 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]=2!

www.vtucs.com

Bellman-Ford Algorithm

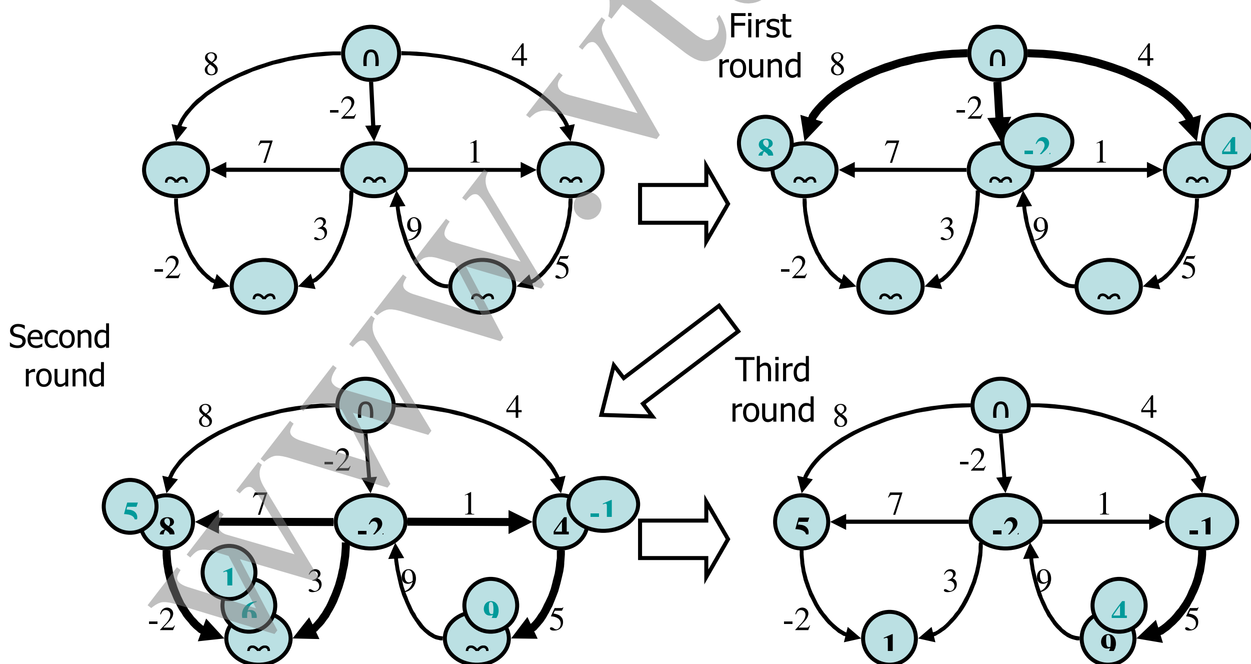
- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists
  - How?

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

www.vtucs.com

Bellman-Ford Example

Nodes are labeled with their d(v) values

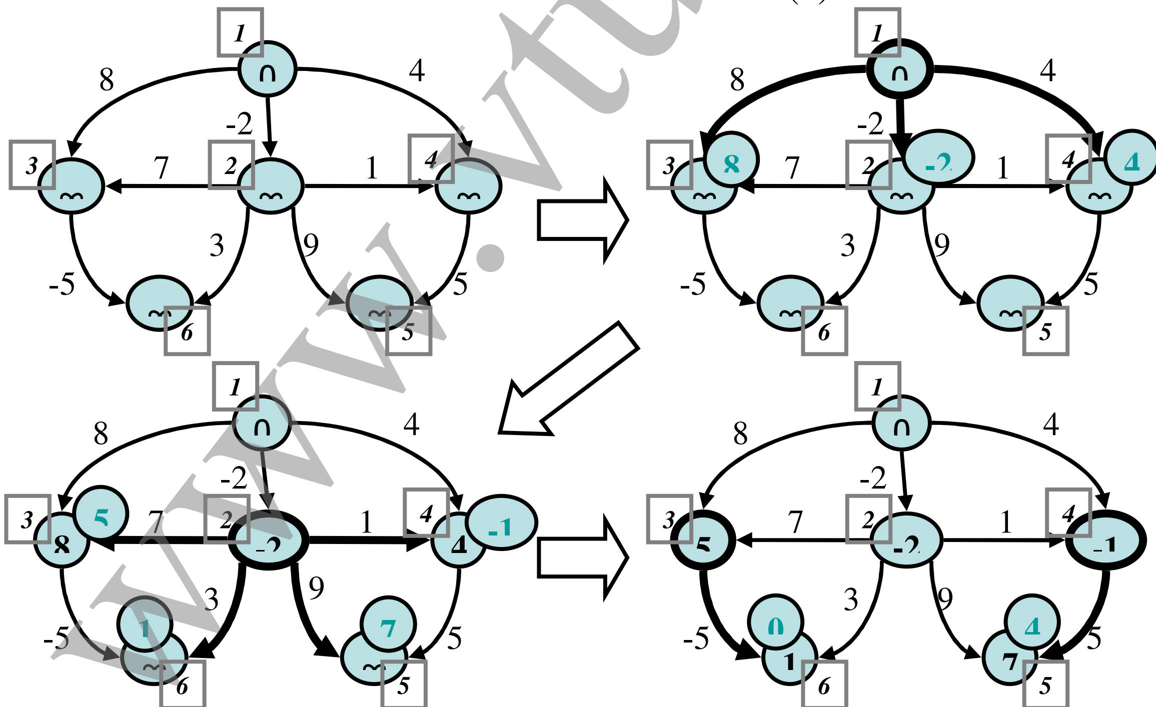


DAG-based Algorithm

- Works even with negative-weight edges
- Uses topological order
- Is much faster than Dijkstra's algorithm
- Running time: O(n+m).

```
Algorithm DagDistances(G, s)
    for all v ∈ G.vertices()
        if v · · s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    Perform a topological sort of the vertices
    for u ← 1 to n do    {in topological order}
    for each e ∈ G.outEdges(u)
        { relax edge e }
        z ← G.opposite(u,e)
        r ← getDistance(u) · · weight(e)
        if r · · getDistance(z)
            setDistance(z,r)
```

DAG Example
Nodes are labeled with their d(v) values

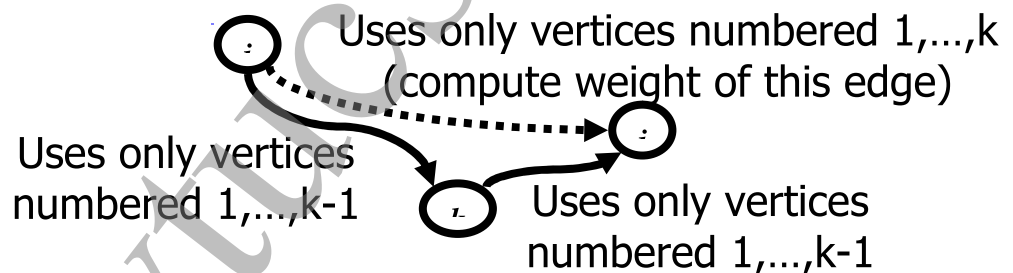


www.vtucs.com

All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes O(nmlog n) time.
- Likewise, n calls to Bellman-Ford would take $O(n^2m)$ time.
- We can achieve $O(n^3)$ time using dynamic programming (similar to the Floyd-Warshall algorithm).

Algorithm *AllPair*(*G*) {assumes vertices 1,…,*n*}
 for all *vertex pairs (i,j)*
    if *i* · · *j*
        $D_0[i,i] \leftarrow 0$
    else if *(i,j) is an edge in G*
        $D_0[i,j] \leftarrow$ *weight of edge (i,j)*
    else
        $D_0[i,j] \leftarrow +\infty$
 for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do**
          $D_k[i,j] \leftarrow \min\{D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]\}$
return D_n



Dijkstra's algorithm

- S = {1}
- for i = 2 to n do D[i] = C[1,i] if there is an edge from 1 to i, infinity otherwise
- for i = 1 to n-1
  { choose a vertex w in V-S such that D[w] is min
  add w to S (where S is the set of visited nodes)
  for each vertex v in V-S do
  D[v] = min(D[v], D[w]+c[w,v])
  }

  Where |V| = n

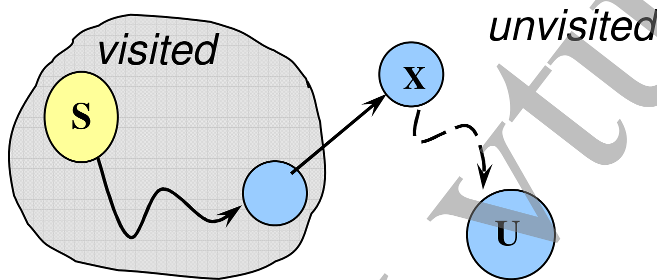
www.vtucs.com

Features of Dijkstra's Algorithm

- A *greedy* algorithm
- "Visits" every vertex only once, when it becomes the vertex with minimal distance amongst those still in the priority queue
- Distances may be revised **multiple times**: current values represent 'best guess' based on our observations so far
- Once a vertex is visited we are guaranteed to have found the shortest path to that vertex…. ***why?***

Correctness (via contradiction)

- Prove D(u) represent the shortest path to u (visited node)
- Assume ***u*** is the first vertex visited such that ***D***(***u***) is ***not*** a shortest path (thus the true shortest path to u must pass through some unvisited vertex)
- Let ***x*** represent the **first unvisited vertex** on the true shortest path to ***u***



- ***D***(***x***) must represent a shortest path to ***x***, and ***D***(***x***) ≤ ***D****shortest*(***u***).
- **However**, Dijkstra's always visits the vertex with the ***smallest distance*** next, so we can't possibly visit ***u*** before we visit ***x***

The Bellman-Ford algorithm

Returns a boolean:

- TRUE if and only if there is no *negative-weight cycle* reachable from the source: a simple cycle $<v_0, v_1,\ldots,v_k>$, where $v_0=v_k$ and

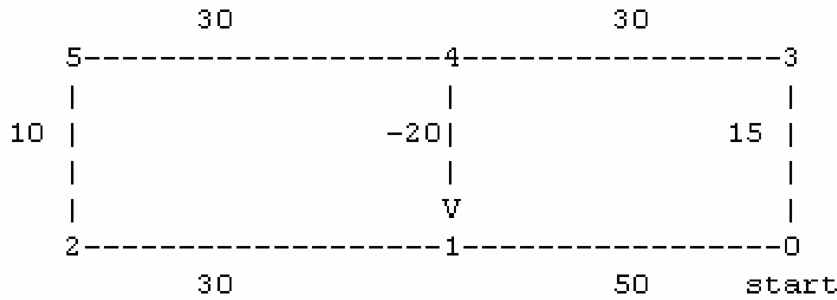
$$\sum_{i=1}^{k} weight(v_{i-1}, v_i) < 0$$

- FALSE otherwise

If it returns TRUE, it also produces the shortest paths

www.vtucs.com

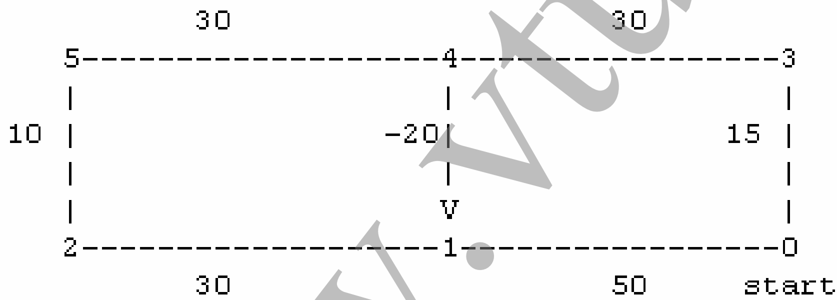
Example



- For each edge (u,v), let's denote its length by (u,v))

- Let d[i][v] = distance from start to v using the shortest path out of all those that use i or fewer edges, or infinity if you can't get there with <= i edges.

  - **How can we fill out the rows?**



V

i

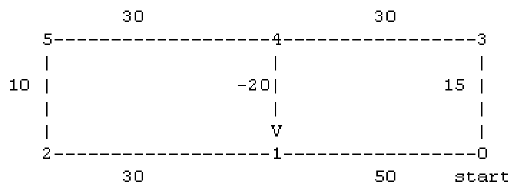
	0	1	2	3	4	5
0	0	∞	∞	∞	∞	∞
1	0	50	∞	15	∞	∞
2	0	50	80	15	45	∞

www.vtucs.com

- Can we get ith row from i-1th row?
- for v != start,
```
 d[v][i] = MIN d[x][i-1] + len(x,v)
       x->v
```
- We know minimum path to come to x using < i nodes.So for all x that can reach v, find the minimum such sum (in blue) among all x
- Assume d[start][i] = 0 for all i

Completing the table

```
 d[v][i] = MIN d[x][i-1] + len(x,v)
          x->v
```



	0	1	2	3	4	5
0	0	∞	∞	∞	∞	∞
1	0	50	∞	15	∞	∞
2	0	50	80	15	45	∞
3	0	25	80	15	45	75
4	0	25	55	15	45	75
5	0	25	55	15	45	65

Key features

- If the graph contains no negative-weight cycles reachable from the source vertex, after **|*V*| - 1** iterations all distance estimates represent shortest paths…***why?***

www.vtucs.com

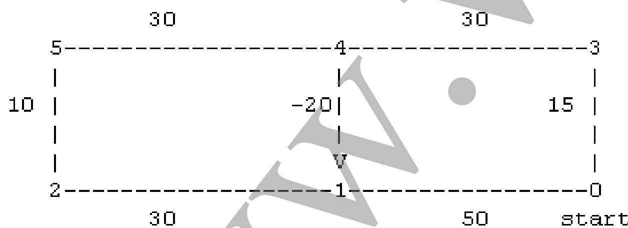
The All Pairs Shortest Path Algorithm (Floyd's Algorithm)

Finding all pairs shortest paths

- Assume G=(V,E) is a graph such that c[v,w] ≥ 0, where C is the matrix of edge costs.
- Find for each pair (v,w), the shortest path from v to w. That is, find the matrix of shortest paths
- Certainly this is a generalization of Dijkstra's.
- Note: For later discussions assume |V| = n and |E| = m

Floyd's Algorithm

```
• A[i][j]  =  C(i,j) if there is an edge (i,j)
• A[i][j]  =  infinity(inf) if there is no edge (i,j)
```



Graph

A =

	0	1	2	3	4	5
0	0	50	inf	15	inf	inf
1	50	0	30	inf	inf	inf
2	inf	30	0	inf	inf	10
3	15	inf	inf	0	30	inf
4	inf	-20	inf	30	0	30
5	inf	inf	10	inf	30	0

"adjacency" matrix

A is the shortest path matrix that uses 1 or fewer edges

www.vtucs.com

- To find shortest paths that uses 2 or fewer edges find A^2, where multiplication defined as *min of sums* instead sum of products
- That is $(A^2)_{ij} = \min\{ A_{ik} + A_{kj} \mid k = 1..n\}$
- This operation is $O(n^3)$
- Using A^2 you can find A^4 and then A^8 and so on
- Therefore to find A^n we need log n operations
- Therefore this algorithm is $O(\log n * n^3)$
- We will consider another algorithm next

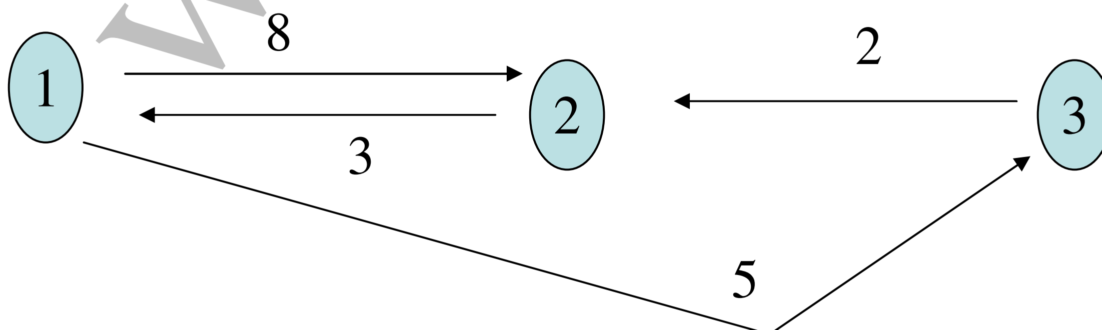
Floyd-Warshall Algorithm

- This algorithm uses nxn matrix A to compute the lengths of the shortest paths using a dynamic programming technique.
- Let A[i,j] = c[i,j] for all i,j & i≠j
- If (i,j) is not an edge, set A[i,j]=infinity and A[i,i]=0
- $A_k[i,j] =$
  $\min(A_{k-1}[i,j],\ A_{k-1}[i,k] + A_{k-1}[k,j])$

Where A_k is the matrix after k-th iteration and path from i to j does not pass through a vertex higher than k

Example – Floyd-Warshall Algorithm

Find the all pairs shortest path matrix



www.vtucs.com

■ $A_k[i,j] =$
$\min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])$

Where A_k is the matrix after k-th iteration and path from i to j does not pass through a vertex higher than k

Floyd-Warshall Implementation

- initialize A[i,j] = C[i,j]
- initialize all A[i,i] = 0
- for k from 1 to n

```
for k from 1 to n
  for i from 1 to n
    for j from 1 to n
      if (A[i,j] > A[i,k]+A[k,j])
        A[i,j] = A[i,k]+A[k,j];
```

- The complexity of this algorithm is $O(n^3)$

Single-Source Shortest Paths

- For a weighted graph $G = (V,E,w)$, the *single-source shortest paths* problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in V.
- Dijkstra's algorithm is similar to Prim's algorithm. It maintains a set of nodes for which the shortest paths are known.
- It grows this set based on the node closest to source using one of the nodes in the current shortest path set.

All-Pairs Shortest Paths

- Given a weighted graph $G(V,E,w)$, the *all-pairs shortest paths* problem is to find the shortest paths between all pairs of vertices $v_i, v_j \in V$.
- A number of algorithms are known for solving this problem.

www.vtucs.com

Floyd's Algorithm

From our observations, the following recurrence relation follows:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\} & \text{if } k \geq 1 \end{cases}$$

This equation must be computed for each pair of nodes and for $k = 1, n$. The serial complexity is $O(n^3)$.

Transitive Closure

- If $G = (V,E)$ is a graph, then the *transitive closure* of G is defined as the graph $G^* = (V,E^*)$, where $E^* = \{(v_i, v_j) \mid \text{there is a path from } v_i \text{ to } v_j \text{ in } G\}$
- The *connectivity matrix* of G is a matrix $A^* = (a_i^*,_j)$ such that $a_i^*,_j = 1$ if there is a path from v_i to v_j or $i = j$, and $a_i^*,_j = \infty$ otherwise.
- To compute A^* we assign a weight of 1 to each edge of E and use any of the all-pairs shortest paths algorithms on this weighted graph.

Johnson's Algorithm

```
1.   procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
2.   begin
3.       Q := V;
4.       for all v ∈ Q do
5.           l[v] := ∞;
6.       l[s] := 0;
7.       while Q ≠ ∅ do
8.       begin
9.           u := extract_min(Q);
10.          for each v ∈ Adj[u] do
11.              if v ∈ Q and l[u] + w(u, v) < l[v] then
12.                  l[v] := l[u] + w(u, v);
13.      endwhile
14.  end JOHNSON_SINGLE_SOURCE_SP
```

Johnson's sequential single-source shortest paths algorithm.

Dijkstra's "shortest path" algorithm

www.vtucs.com

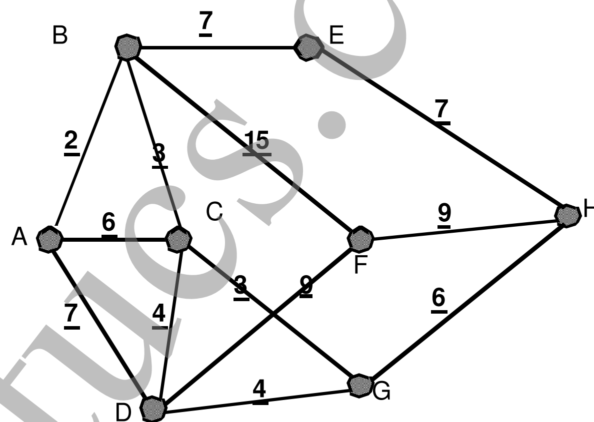
(for non-negative edge weights only)

Key concepts: “labelling”, “scanning”

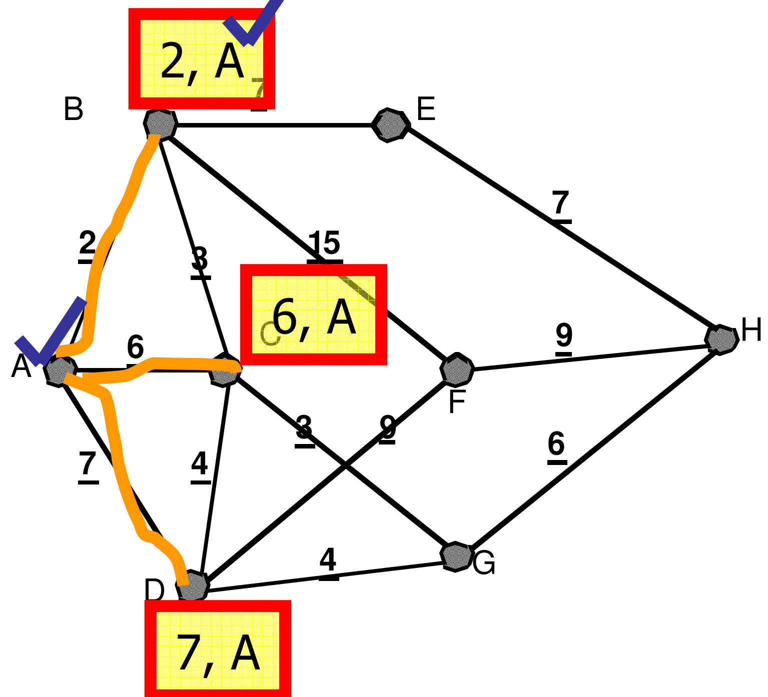
Label = {distance, predecessor}.
Labels are initially “temporary” later “permanent” .

“Scanning” is the process of looking out from a given node to all adjacent nodes that are not permanently labelled and is shorter.

<u>Example:</u>
find shortest A-H route if the edge weights represent distance



<u>Start:</u> [P] a vector of permanently labelled nodes, [D] a vector of distances
- [P] <- [A]
- All other nodes are unlabelled
- [D] <- all infinite
- source is already perm. labelled



<u>Step 1: “Scan” from node A:</u>

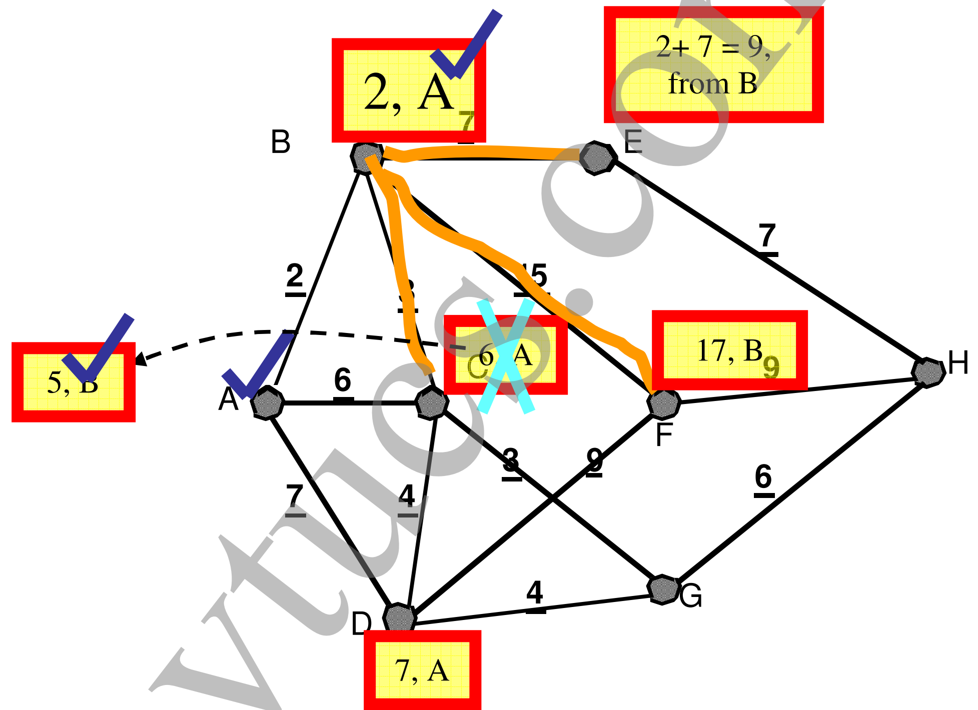
- nodes B, C, D get temp labels
- node B gets perm label:
- [P] = [A, B]
- [D] = [-, 2]

www.vtues.com

Step 2: "Scan" from node B:

- scan goes only to other nodes not yet perm labelled
- node E gets {9, from B}
- node F gets {17, from B}
- node C gets its label *updated:* {5, from B}
- node C is lowest global distance so it gets the permanent label this time

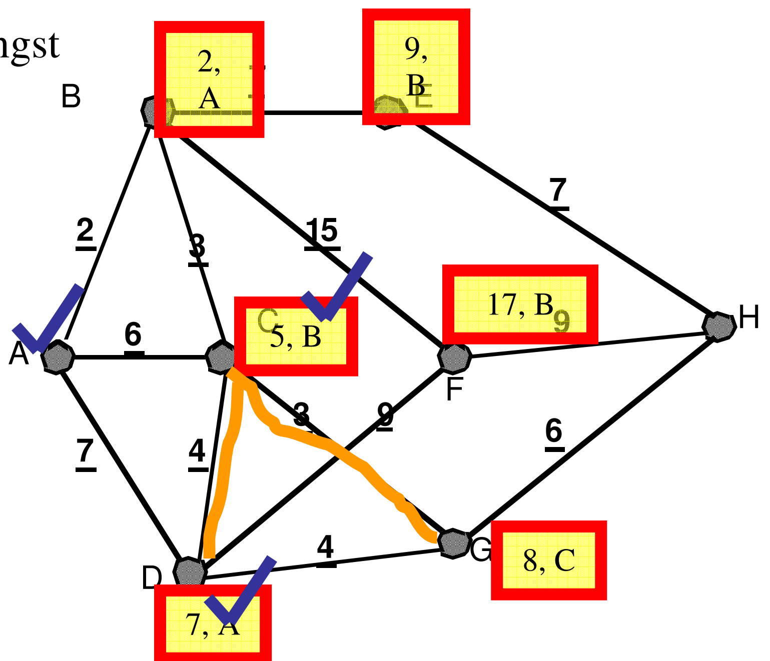
- [P] = [A, B, C]
- [D] = [-, 2, 5]



Step 3: "Scan" from node C:

- node G gets {8, from C}
- node D *keeps* {7, from A}
- node D is lowest global distance amongst temp labels so it gets the permanent label this time

- [P] = [A, B, C, D]
- [D] = [-, 2, 5, 7]



www.vtuc.com

Step 4: “Scan” from node D:

- node F gets *updated* label {16, from D}
- node G *keeps* {8, from C}
- node G is lowest global distance amongst temp labels so it gets the permanent label this time

- [P] = [A, B, C, D, G]
- [D] = [-, 2, 5, 7, 8]

Step 5: “Scan” from node G:

- node H gets label {14, from G}
- Since H is also target tempting to stop, *but not yet.*
- Node E (not *just* scanned) gets next perm. Label.
- [P] = [A, B, C, D, G, E]
- [D] = [-, 2, 5, 7, 8, 9]

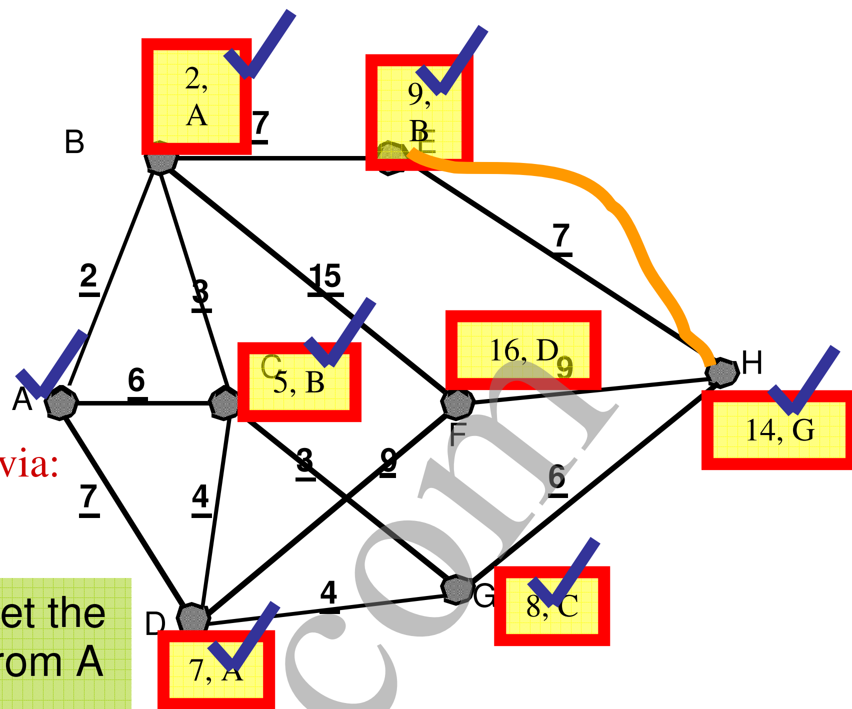
www.vtucs.com

<u>Step 6: "Scan" from node E:</u>

- node H *keeps* label {14, from G}
- node H label permanent now
- [P] = [A, B, C, D, G, E, H]
- [D] = [-, 2, 5, 7, 8, 9, 14]

hence:

- A-H shortest route is of distance 14, via: H -> G -> C -> B -> A

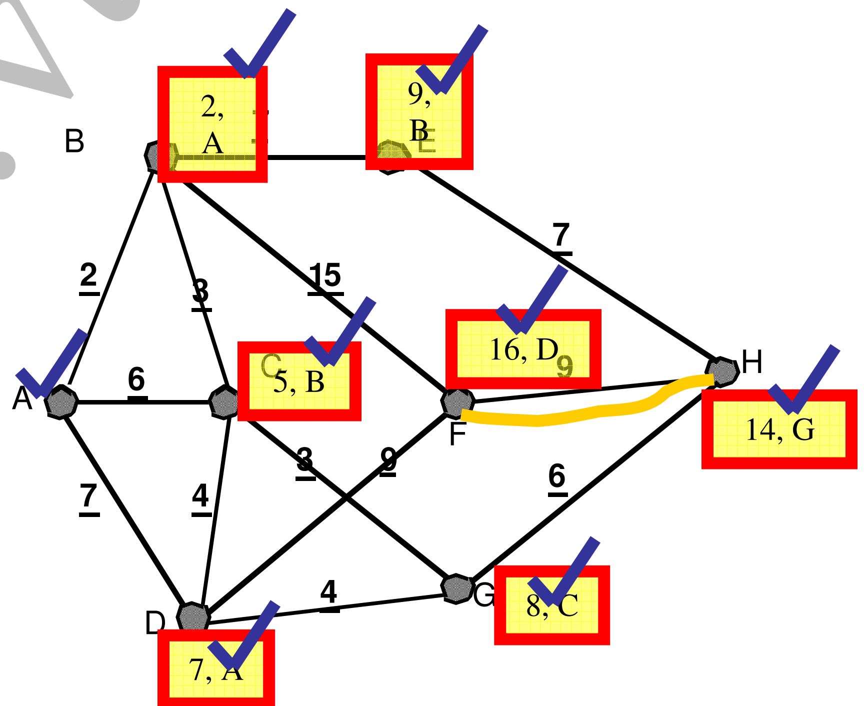


If we continue, however, we will get the complete tree of shortest routes from A to all other nodes...

Finally all nodes have permanent labels... the complete tree of shortest routes from A to all other nodes has been found...

<u>Step 7: only node F is not permanently labelled:</u>

- scan from H
- Node F label {16, from D} is made permanent
- [P] = [A, B, C, D, G, E, H, F]
- [D] = [-, 2, 5, 7, 8, 9, 14, 16]



www.vtucs.com

Recap:

start at source (first perm label):

scan from latest perm label,
update temp labels
find lowest temp label
make perm

loop till: target is perm. label (path only)
all nodes perm (whole tree)

Things to remember:

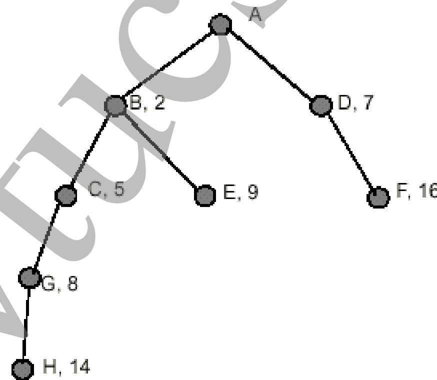
(1) Only stop when target node is *permanently* labelled.

(2) The next node to scan from *may not* be one just visited in the last scan.

At the end, the [D], [P] vectors (plus labels) encode the shortest path tree:

Table 2-1

step	node	perm. label
0	A	(source)
1	B	{2, from A}
2	C	{5, from B}
3	D	{7, from A}
4	G	{8, from C}
5	H	{14, from G}
6	E	{9, from B}
	F	{16, from D}



Motivation

- Given a connected, positive weighted graph
- Find the length of a shortest path from vertex a to vertex z.

www.vtucs.com

Dijkstra's Algorithm

- Input: A connected, positive weighted graph,vertices a and z
- Output: L(z), the length of a shortest path from a to z

```
1.  Dijkstra(w,a,z,L){
2.      L(a)=0
3.      for all vertices x ≠a
4.            L(x)= ∞
5.      T=set of all vertices
6.       while(z ϵ T){
7.             choose v ϵ T with minimum L(v)
8.             T=T-{v}
9.             for each x ϵ T adjacent to v
10.                    L(x)=min{L(x),L(v)+w(v,x)}
11.          }
12.    }
```

Proof of Dijkstra's Algorithm

- Basic Step(i=1):
  we set L(a)=0, and L(a) is sure the length of a shortest path from a to a.

- Inductive step: For an arbitrary step i
  Suppose for step $k<i$, L(v) is the length of a shortest path from a to v.
  Next, suppose that at the it step we choose v in T with minimum L(v).
  We will seek a contradiction that if there is a w whose length is less than L(v) then w is not in T.

By way of contradiction, suppose there is a w with $L(w)<L(v)$, $w \epsilon T$. Then, let P be the shortest path from a to w, and let x be the vertex nearest to a on P that is in T and let u be x's predecessor. The node u must not be in T (because x was the nearest node to a that was in T). By assumption, L(u) was the length of the shortest path from a to u.

Then, $L(x) \leq L(u)+w(u,x) \leq$ length of $P < L(v)$. This is a contradiction. So w is not in T. According to our assumption, every path from a to v has length at least L(v).

www.vtucs.com

www.vtucs.com