GRAPH THEORY and APPLICATIONS

Shortest Paths

Shortest Path

- Weighted digraph: A directed graph with real valued weights assigned to each edge.
  - G(V,E,w)
- Length of a path in a weighted digraph: Sum of the lengths of the edges on the path.
- Shortest path: A path between two nodes of least length.

Dijkstra's Method

- Let G(V,E) be a weighted digraph all of whose edge weights are positive.
- x and y are vertices of G.

Aim: Find the shortest path from x to y and its length, or show there is none.

- The method uses a search tree technique based on:
  - k^{th} nearest vertex to x is the neighbor of one of the j^{th} nearest vertices to x for some $j < k$.

Dijkstra's Method

- Let:
  - `Near(j)` denote the j^{th} nearest vertex to x
  - `Dist(u)` distance from x to any vertex u
  - `Length(u,v)` edge length from u to any neighbor v.
- Then, the k^{th} nearest vertex to x is v that minimizes:

  `Dist(Near(j)) + Length(Near(j),v)`

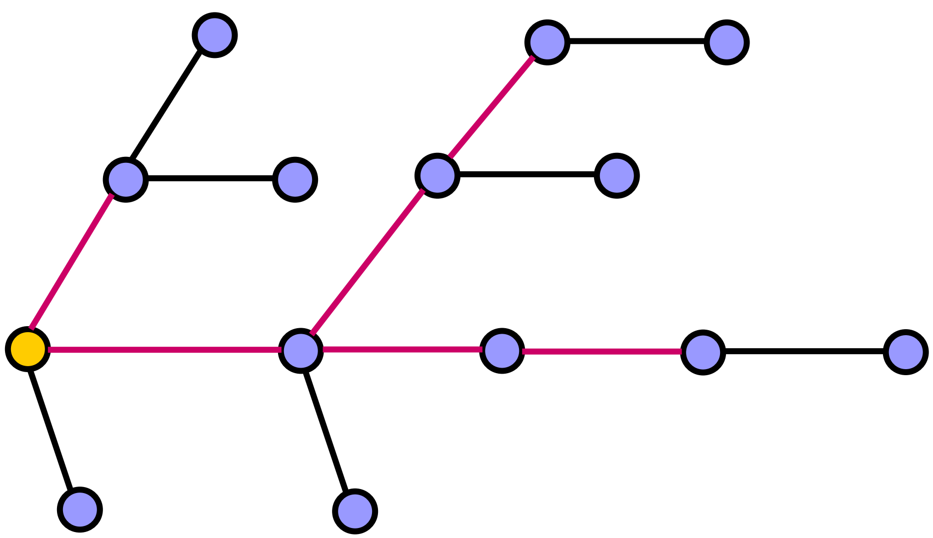
  where the minimum is taken over all $j < k$.
- So, to find the distance to y, we first find the distances to all vertices closer to x than y.

Dijkstra's Method

- Successively more distant vertices from x are found using a search procedure which explores the graph in a tree-like manner.
- This search induces a subgraph of G called a search tree.
- This tree contains a subtree called a shortest path subtree.
- At each phase, a new vertex v lying in the search tree is explored, and the search tree is extended from v to its neighbors.

Dijkstra's Method

- Initially, the search tree fans out from x to its immediate neighbors.
- After k stages, the shortest path subtree of the search tree contains the k nearest vertices to x.
  - The path through this tree from x to any of its vertices is a shortest path.



• **Black Edges**:
lead to vertices of the search tree, but not yet in the shortest path subtree.
• **Pink Edges**:
lead to vertices which are in the shortest path subtree.
• **Any edge not shown**:
  - unexplored
  - Don't to lie on the shortest path

Dijkstra's Algorithm

- Function Dijkstra (G, x, y)
  - Returns the shortest distance from x to y in Dist[y]
  - Returns the shortest path using the Pred field starting at y
  - or fails.
- Dist[0..|V|]: real
  - Contains the current estimated distance to v from x.
- Pred[0..|V|]: 0..|V|
  - Gives the index of the search tree predecessor of v.

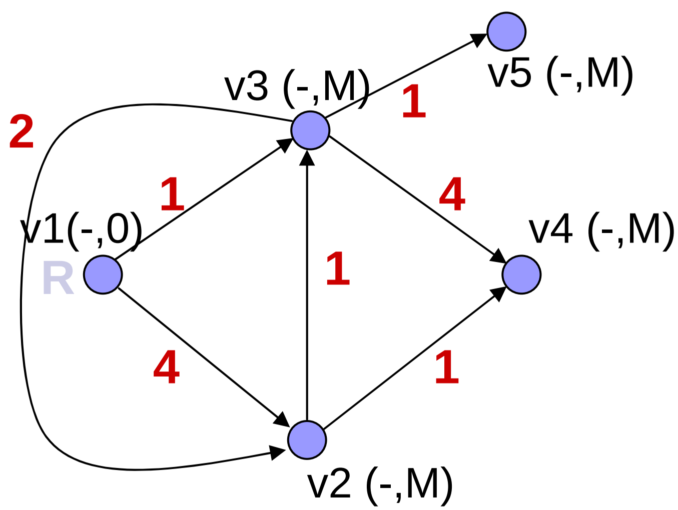
Function Dijkstra

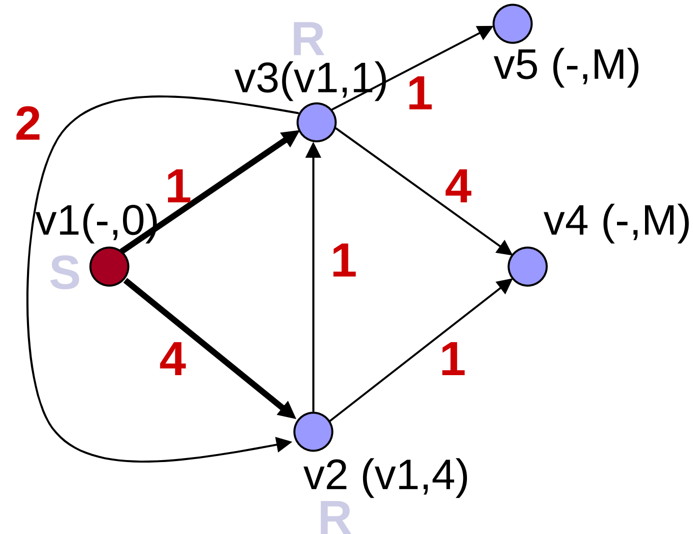
```
Reached = {x}
Pred(w) = 0 for each vertex w in G
Dist(x) = 0
Dist(w) = M, for each w <> x
while getmin(v) and v <> y do
  for each neighbor w of v do
    if w unreached then
      add w to Reached
      Dist(w) = Dist(v) + Length(v,w)
      Pred(w) = v
    else
      if w in Reached and Dist(w) > Dist(v) +
                                    Length(v,w) then
        Dist(w) = Dist(v) + Length(v,w)
        Pred(w) = v
Dijkstra = (v = y)
```

getmin(v):
- returns the vertex v in Reached with the minimum value of Dist(v)
- removes v from Reached
- places v in shortest path tree

Example

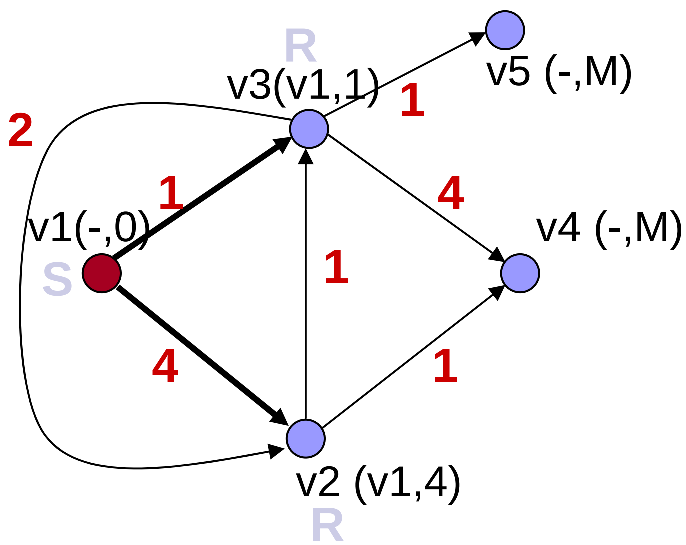
The shortest path from v1 to v4 is sought.



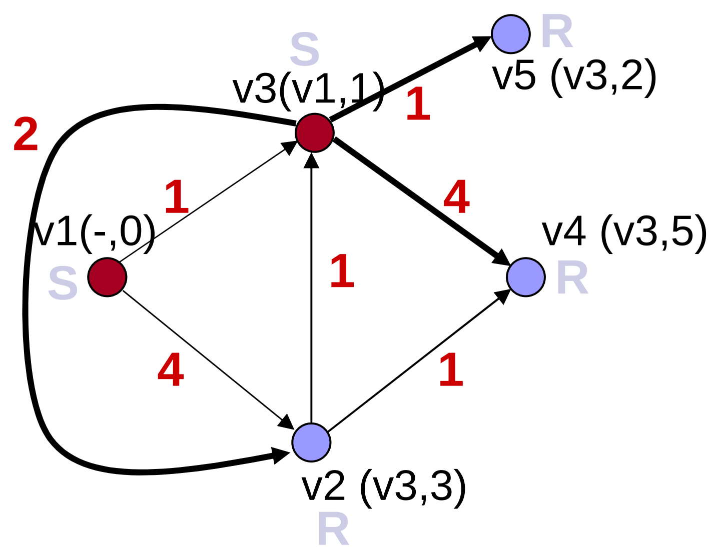


Weighted digraph G

Example

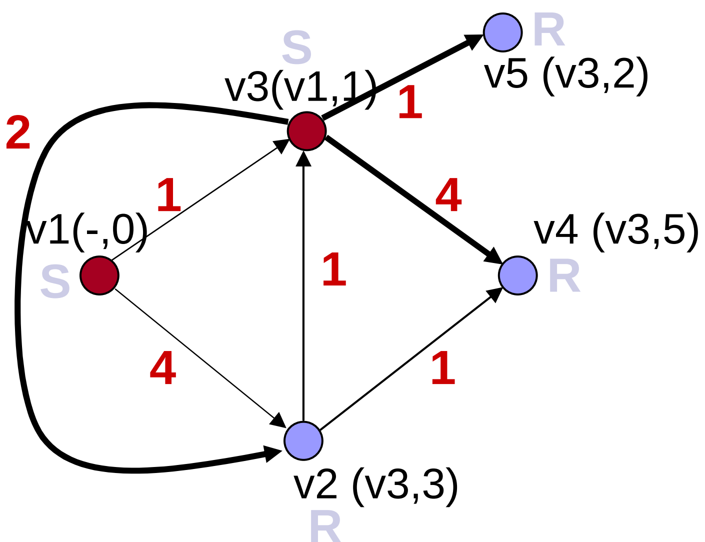


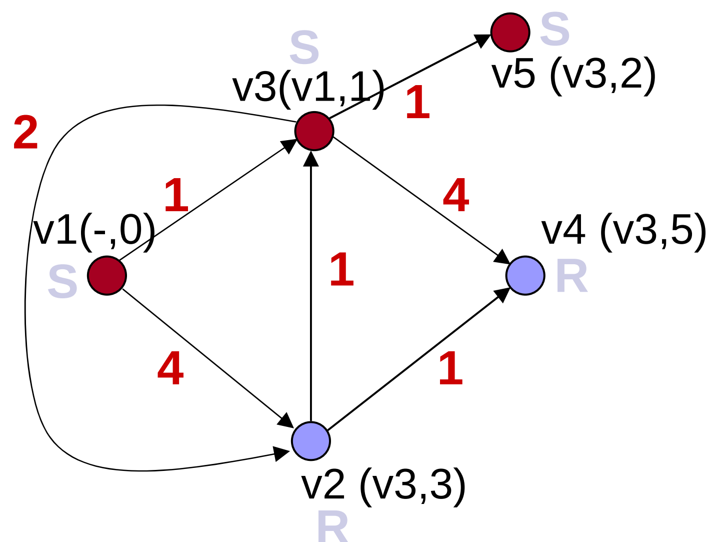
R
v3(v1,1)
v5 (-,M)
2
1
1
4
v1(-,0)
S
v4 (-,M)
1
4
1
v2 (v1,4)
R



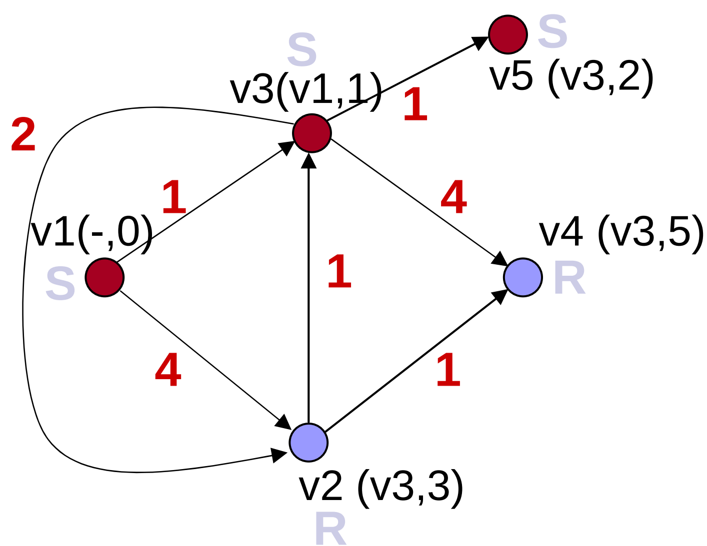
S
R
v3(v1,1)
v5 (v3,2)
2
1
1
4
v1(-,0)
S
v4 (v3,5)
R
1
4
1
v2 (v3,3)
R

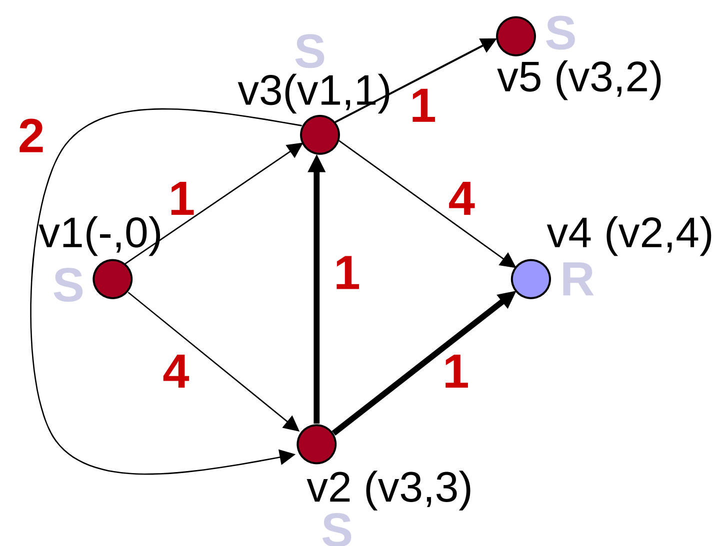
Example


R
S
v3(v1,1)
v5 (v3,2)
1
2
1
4
v1(-,0)
v4 (v3,5)
S
1
R
4
1
v2 (v3,3)
R


S
S
v3(v1,1)
v5 (v3,2)
1
2
1
4
v1(-,0)
v4 (v3,5)
S
1
R
4
1
v2 (v3,3)
R

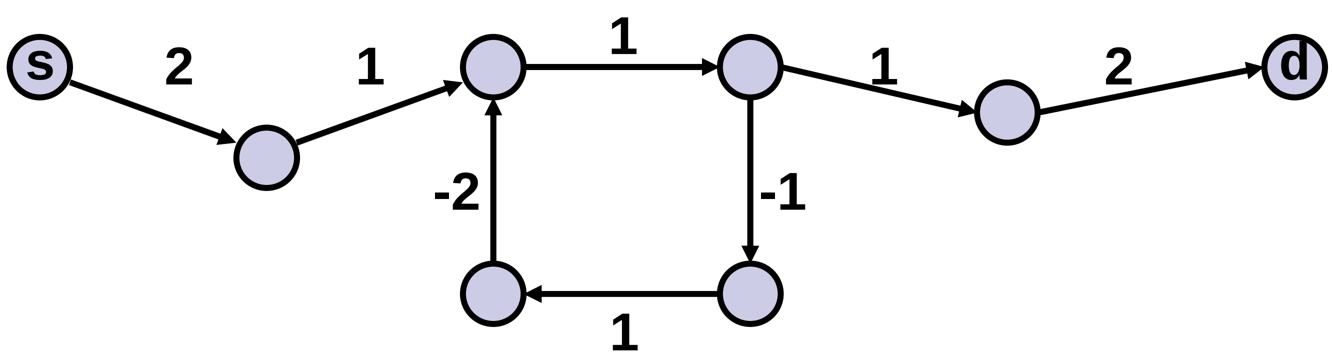
Example


S
S
v3(v1,1)
v5 (v3,2)
1
2
1
4
v1(-,0)
v4 (v3,5)
S
1
R
4
1
v2 (v3,3)
R


S
S
v3(v1,1)
v5 (v3,2)
1
2
1
4
v1(-,0)
v4 (v2,4)
S
1
R
4
1
v2 (v3,3)
S

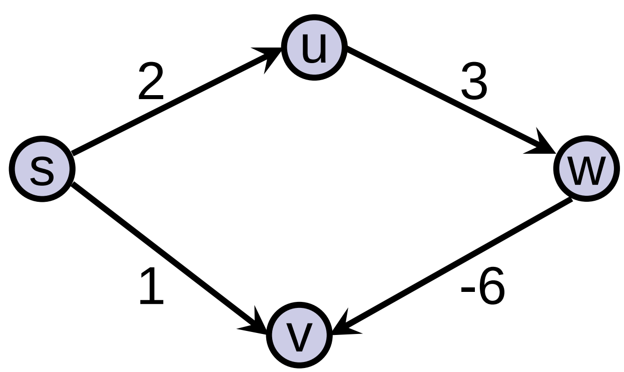
Negative Cycles

- Shortest path problem is considered under the assumption that there are no negative cycle in the graph.
- If there is a negative cycle C:
  - Path P_s from source to C
  - Go around C as many times as you want
  - Path P_d from cycle to destination



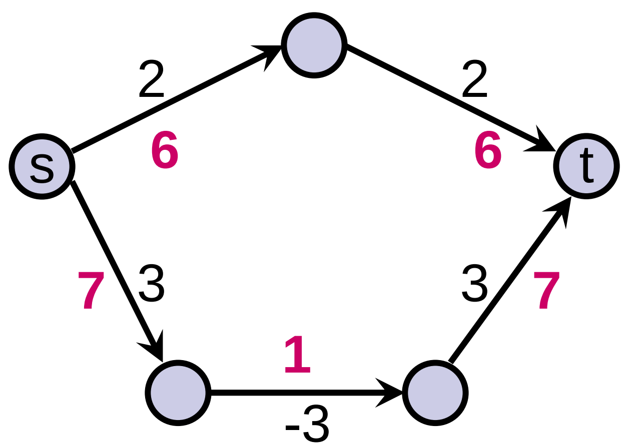
Why Dijkstra don't work with negative cycles

- Start with S = {s}
- Minimum cost path leaving s is (s,v): Add v to S
- Shortest path from s to v is (s,v) assuming there are no negative weighted edges.
- But, this is no longer true:
  - Minimum length path from s to v: s-u-w-v



Can we modify costs?

- A natural idea:
  - Modify costs by adding some large constant M
  - $c_{ik}^{new} = c_{ik} + M$ for each edge
  - M is large enough, all c_{ik}^{new} are positive.
  - Then, use Dijkstra's method.



- Changing costs changes the minimum cost paths.
- We added:
  - 2M to upper path
  - 3M to the lower path

Floyd's Algorithm: All Vertex Pairs

- Floyd's algorithm allows negative edge weights.
- It finds shortest paths between every pair of vertices in G.
- Provides a matrix representation for the $|V|^2$ shortest paths found.

Floyd's Method

- Dynamic programming is used.
- At stage k, we have:
  - the shortest paths, and
  - distances

  between every pair of vertices, where the internal vertices have indices on 0..k
- We progress from the solutions at stage k to the solutions at stage k+1, by allowing k+1 as an intermediate vertex if it improves the current distances.

Floyd's Algorithm

- The graph is represented by its distance matrix `Dist`.
  - **`Dist(i,j)`** gives the length of the (i,j) edge.
  - Diagonal set to 0.
  - If there is no edge between (i,j), set to some large positive number M.
  - Stage k shortest distances are in a |V|x|V|x(|V|+1) array SD(i,j,k).
  - The outermost for loop is indexed by the stage k.

Procedure Floyd

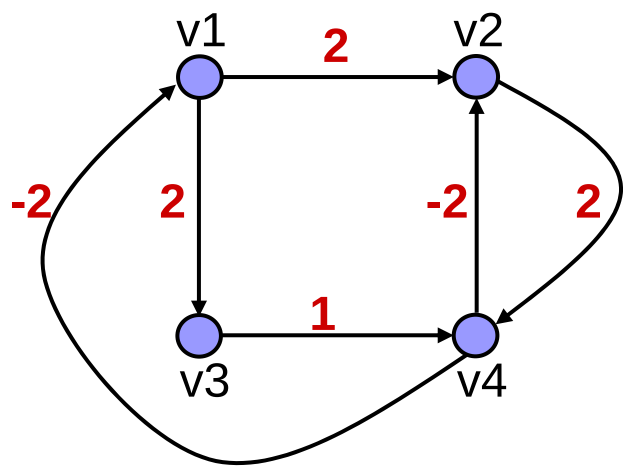
```
SD(1..|V|, 1..|V|, 0..|V|) : Real

for i,j = 1..|V| do
  SD[i,j,0] = Dist[i,j]

for k = 1..|V| do
  for i = 1..|V| do
    for j = 1..|V| do
      SD[i,j,k] = min{SD[i,j,k-1],
                      SD[i,k,k-1] + SD[k,j,k-1]}
```

- A refinement is needed to find and store the shortest paths.

Procedure Floyd_paths

```
SD(1..|V|, 1..|V|, 0..|V|) : Real
SP(1..|V|, 1..|V|, 0..|V|) : 1..|V|
for i,j = 1..|V| do
  SD[i,j,0] = Dist[i,j]
  SP[i,j,0] = j
for k = 1..|V| do
  for i = 1..|V| do
    for j = 1..|V| do
      if SD[i,j,k-1] < SD[i,k,k-1] + SD[k,j,k-1]} then
        SD[i,j,k] = SD[i,j,k-1]
        SP[i,j,k] = SP[i,j,k-1]
      else
        SD[i,j,k] = SD[i,k,k-1] + SD[k,j,k-1]
        SP[i,j,k] = SP[i,k,k-1]
      endif
```

Example



SD

	v1	v2	v3	v4
v1	0	2	2	M
v2	M	0	M	2
v3	M	M	0	1
v4	-2	-2	M	0

SP

	v1	v2	v3	v4
v1	1	2	3	4
v2	1	2	3	4
v3	1	2	3	4
v4	1	2	3	4

Example k=1

	v1	v2	v3	v4
v1	0	2	2	M
v2	M	0	M	2
v3	M	M	0	1
v4	-2	-2	M	0

	v1	v2	v3	v4
v1	1	2	3	4
v2	1	2	3	4
v3	1	2	3	4
v4	1	2	3	4

	v1	v2	v3	v4
v1	0	2	2	M
v2	M	0	M	2
v3	M	M	0	1
v4	-2	-2	0	0

	v1	v2	v3	v4
v1	1	2	3	4
v2	1	2	3	4
v3	1	2	3	4
v4	1	2	1	4

Example k=2

	v1	v2	v3	v4
v1	0	2	2	M
v2	M	0	M	2
v3	M	M	0	1
v4	-2	-2	0	0

	v1	v2	v3	v4
v1	1	2	3	4
v2	1	2	3	4
v3	1	2	3	4
v4	1	2	1	4

	v1	v2	v3	v4
v1	0	2	2	4
v2	M	0	M	2
v3	M	M	0	1
v4	-2	-2	0	0

	v1	v2	v3	v4
v1	1	2	3	2
v2	1	2	3	4
v3	1	2	3	4
v4	1	2	1	4

Example k=3

	v1	v2	v3	v4
v1	0	2	2	4
v2	M	0	M	2
v3	M	M	0	1
v4	-2	-2	0	0

	v1	v2	v3	v4
v1	1	2	3	2
v2	1	2	3	4
v3	1	2	3	4
v4	1	2	1	4

	v1	v2	v3	v4
v1	0	2	2	3
v2	M	0	M	2
v3	M	M	0	1
v4	-2	-2	0	0

	v1	v2	v3	v4
v1	1	2	3	3
v2	1	2	3	4
v3	1	2	3	4
v4	1	2	1	4

Example k=4

	v1	v2	v3	v4
v1	0	2	2	3
v2	M	0	M	2
v3	M	M	0	1
v4	-2	-2	0	0

	v1	v2	v3	v4
v1	1	2	3	3
v2	1	2	3	4
v3	1	2	3	4
v4	1	2	1	4

	v1	v2	v3	v4
v1	0	1	2	3
v2	0	0	2	2
v3	-1	-1	0	1
v4	-2	-2	0	0

	v1	v2	v3	v4
v1	1	3	3	3
v2	4	2	4	4
v3	4	4	3	4
v4	1	2	1	4

Extracting the Shortest Paths

- This procedure returns the shortest path from i to j in array P.
- P is initially set to 0.

```
Procedure Extract_shortest_path (SP,|V|,i,j,P)

P[0] = i
k = i
cnt = 1
while k <> j do
  k = SP[k,j,|V|]
  P[cnt] = k
  cnt++
```

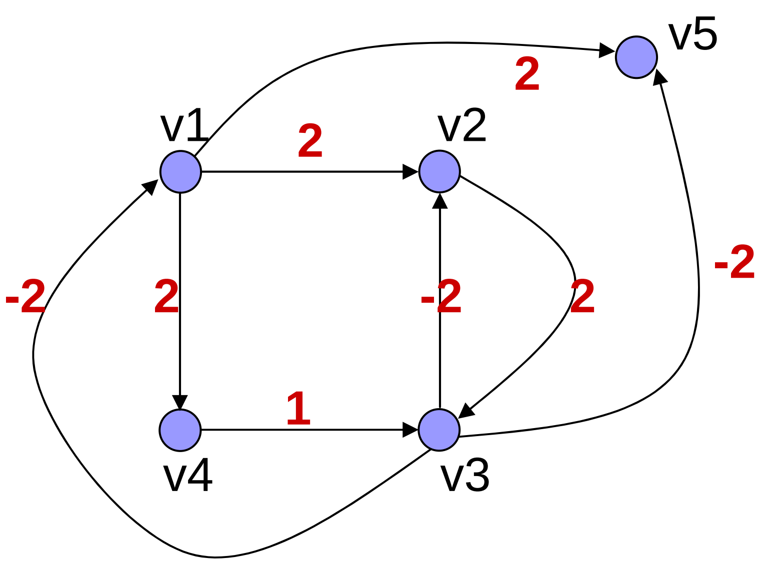
Ford's Algorithm: Vertex to All Vertices

- Also called Bellman-Ford
- Finds the shortest paths from a vertex v to every vertex.
- By the end of k^{th} iteration the algorithm finds all the shortest paths emanating from v that have at most k edges.
- We maintain a predecessor pointer for each vertex u.
  - It points to the predecessor of u on the current best shortest path from v to u: Pred(u).
- Length(u,v) gives the length of (u,v) edge.
- Dist(u) gives the length of the estimated shortest path to u.

Function Ford

```
Pass = 0
Dist[v] = 0
Dist[u] = M for all u <> v

repeat
  Ford = True
  Pass++
  for every edge (u,w) in G do
    if Dist[u] + Length[u,w] < Dist[w] then
      Dist[w] = Dist[u] + Length[u,w]
      Pred[w] = u
      Ford = False
    endif
until Ford or Pass >= |V|
```

Example



	pass 1		pass 2		pass 3	
(u,w)	Dist (w)	Pred (w)	Dist (w)	Pred (w)	Dist (w)	Pred (w)
1,2	2	1	2	1	1	3
1,4	2	1	2	1	2	1
1,5	2	1	2	1	1	3
2,3	4	2	3	4	3	4
3,1	0	0	0	0	0	0
3,2	2	1	1	3	1	3
3,5	2	1	1	3	1	3
4,3	3	4	3	4	3	4

Detecting negative cycles

- Finding negative cycles in a graph with negative cycles, is NP-complete.
- In fact, label-correcting algorithms, may never terminate.
- How do we detect a graph contains a negative cycle?
- Two facts:
  - A path contains at most $n\text{-}1$ arcs.
  - Assuming C is the maximum edge cost, a path's cost is at least $-nC$.

Detecting negative cycles

- If we find that the distance label of some node k has fallen below $-nC$, we can terminate computation.
- The negative cycle can be obtained by tracing the predecessor indices starting at node k.

Detecting negative cycles

A second method:

- Check at repeated intervals to see whether the predecessor graph (shortest path tree) contains a cycle.
- Predecossor graph is a not a tree
(it contains a cycle)
$\Leftrightarrow$ The graph contains a negative cycle.
- O(n)-time algorithm. Run it every α label updates.

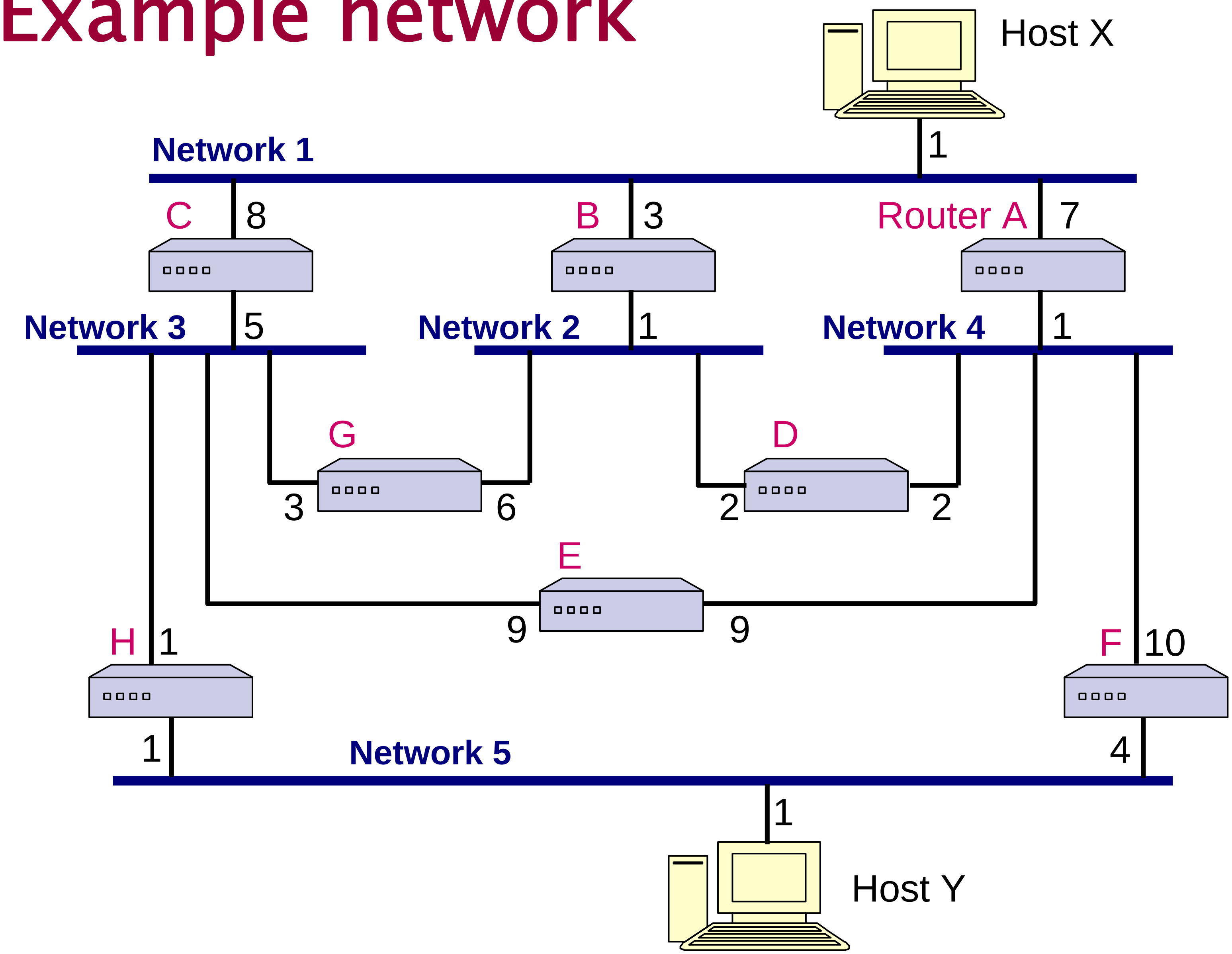
Detecting negative cycles

```
Source is labeled, all other nodes are unlabeled;
for each unlabeled node k do
   Label node k with k;
  current = k;
  repeat
    i = predecessor[current];
    if label[i] == k then
      cycle detected, exit;
    else
      label[i] = k;
    endif
    current = i;
    if (current == source)
      (and predecessor[source] == k) then
      cycle detected;
  until current <> source;
```

Application: Internet Routing

- RIP: Routing Information Protocol (1988)
- A widely used protocol
- Uses a technique known as distance-vector routing
- Each node (router or host) exchange information with its neighbors.

Example network



Distance-vector Routing

Each node x maintains three vectors:

1. Link cost vector:

$$W_x = \begin{bmatrix} w(x,1) \\ \dots \\ w(x,M) \end{bmatrix}$$

M : number of networks to which x directly attaches

$w(x,i)$: output for each attached network

2. Distance vector:

$$L_x = \begin{bmatrix} L(x,1) \\ \dots \\ L(x,N) \end{bmatrix}$$

$L(x,j)$: current estimate of minimum delay to network j

N : number of networks

Distance-vector Routing

3. Next-hop vector:

$$R_x = \begin{bmatrix} R(x,1) \\ \ldots \\ R(x,N) \end{bmatrix}$$

R(x,j) : next router in the current minimum delay route to network j

- Every 30 seconds each node exchanges its distance vector with all of its neighbors.
- Receiving incoming distance vectors, node x updates its vectors.

Distance-vector Routing

- Node x calculates:

$$L(x, j) = \underset{y \in A}{\text{Min}}\, [L(y, j) + w(x, N_{xy})]$$

$$R(x, j) = y \quad y \text{ that minimizes above expression}$$

A : set of neighbors of x

N_{xy} : network connecting x to y

Example: Routing table for host X

Destination network	Next router	L(X,j)
1	-	1
2	B	2
3	B	5
4	A	2
5	A	6

- At some point suppose the link costs change:
  - Both link costs from E become 1
  - Both link costs from F become 1
- Assume that X's neighbors learn of the change.

Example

B	C	A
3	8	6
1	8	3
4	5	2
3	6	1
4	6	2

Delay vectors sent to X from neighbor routers

Routing table of X after update

Destination network	Next router	L(X,j)
1	-	1
2	B	2
3	A	3
4	A	2
5	A	3

Distributed Bellman-Ford Algorithm

- The update calculation of RIP is essentially the same as Bellman-Ford algorithm's.
  - RIP uses a distributed version of Bellman-Ford.
- The algorithm is run in asynchronous mode.
- Each router x begins with:

$$L(x, j) = \begin{cases} w(x, j) & \text{if x is directly connected to network j} \\ \infty & \text{otherwise} \end{cases}$$

- Every 30 second each router transmits its distance vector to its neighbors.
- A router updates its table after receiving new distance vectors from all its neighbors.