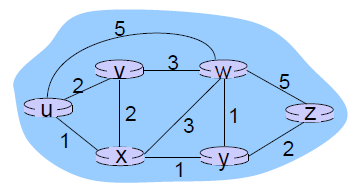
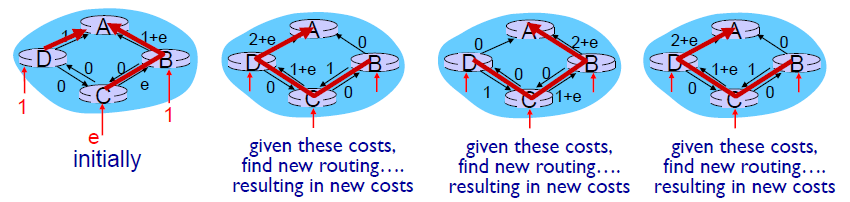
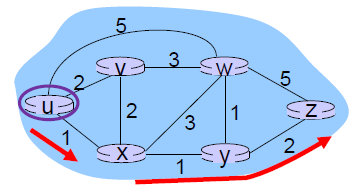
**Computer Network  
Chapter 5: Network Layer: The Control Plane – part 1**

**Routing Protocols  
 Routing Algorithm의 목적:** Source router에서 destination router까지 좋은 경로를 찾기 위함.   
 Default router (first-hop router)는 host와 직접 연결 된 router.  
 Source router: 출발지 host의 default router.  
 Destination router: 목적지 host의 default router.  
 Good path: the least cost.  
 Min-hop path: 라우터를 최소로 거쳐가는 길.  
  
 **Graph abstraction** Graph: G = (N,E)  
 N = set of routers = { u, v, w, x, y, z}  
 E = set of links = { (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) }  
  
각 links마다 cost가 있다.  
 🡪 c(x,x') = cost of link (x, x’),  
 🡪 cost of path = 출발지와 목적지를 연결하는 path들의 cost의 합. 🡺 경로는 여러 개 나올 수 있기 때문에, 그 중 cost가 가장 작은 것을 찾아야한다.  
  
 **Routing algorithm classification:**   
 global: 모든 router (전체 네트워크 정보)를 알고 있을 때 🡪 **link state algorithm**  
 decentralized: 이웃의 정보만 알고 있을 때 🡪 **distance vector algorithm**

**Routing Protocols: Link State  
 Link State Routing:** link state routing은 ‘**globa**l’ 해야한다. 그러기 위해서 (각 라우터들은 이웃 정보를 알고 있다. 라는 가정 – 이웃한 router는 같은 subnet안에 있기 때문에 알 수 있음.) 각 라우터들은 **flooding** (자신의 이웃 정보를 모두에게 보낸다) 한다. flooding을 하면 모든 라우터는 **전체 네트워크 지도**를 알 수 있다. 이 그림을 **routing algorithm**을 한다: 각 node에 대한 least path를 구하고 forwarding table에 저장한다.  
 node A는 Dijkstra’s algorithm을 통해 A에서 모든 목적지까지의 최단 경로를 알 수 있다.  
  
 **Dijkstra’s Algorithm** Input: Network graph, with link costs  
 Output: 한 node (source)에서 다른 모든 노드까지의 최소비용의 path 🡪 forwarding table에 넣어준다.  
 c(i,j): node i에서 j까지의 link cost // indirect node는 무한대로 표시  
 D(v): source에서 destination 현재 v(node)의 cost value  
 p(v): source에서 v까지의 path를 따른 선행노드 (v의 이웃)  
 N’: 최소비용 경로(최소 D(v))가 결정적으로 알려진 node의 subset  
 Complexity: O(n2) // heap structure 일 때는 O(nlogn)  
 Oscillations possible: e.g.) suppport link cost equals amount of carried traffic.  
 Consider path from C to A (clockwise vs. counterclockwise path.  


**Routing Protocols: Distance Vector**  
 **Distance Vector Algorithm:** 점차적으로 이웃으로부터의 정보를 가지고 자신의 DV를 업데이트 해 가는 과정.  
(모든 노드는 각각 이웃한 node이 정보를 가지고 있다. (LS같이)). 각 node들은 Distance vector (DV)를 가지고 있다. – 자신으로부터의 모든 다른 node의 잠정적인 shortest path를 알고있다. **각자 가지고 있는 DV를 이웃에게 보낸다**. 🡪 **이웃은 그 DV를 보고 자신의 DV를 업데이트 한다.   
  
 LS vs. DS**  
 Link State는 이웃의 정보를 모두에게… 🡪 전체 그림을 알 수 있다. 🡺 전체 그림을 보고 Dijkstra’s algorithm을 이용해서 계산  
 Distance Vector는 모두까지의 정보를 이웃한테만… 🡪 이웃의 정보만 받기 때문에 전체 그림을 모른다. 🡺 bellman-ford로 계산 (이웃으로부터 정보를 받으면 bellman-ford 실행)  
  
 **Bellman-Ford equation**: **dx(y) = min { c(x,v) + dv(y) }** x에서 v까지의 cost(이웃한 node의 cost)와 이웃한 node (v)에서 destination (y)까지의 cost중 최소를 찾는 것  
bellman-ford를 했음에도 DV 값에 업데이트가 없으면 최적의 경로를 찾았음을 의미함.  
 example:   
dv(z) = 5, dx(z) = 3, dw(z) = 3 🡪 v에서 z까지의 cost, x에서 z까지의 cost, w에서 z까지의 cos DV를 u가 받고 bellman-ford를 이용해서 du(z) = min { c(u,v) + dv(z) }를 구한다.  
🡺 u에서 이웃한 v까지 가서 v에서 z까지의 cost를 합한 것과, u에서 이웃한 x까지의 코스트와 x에서 z까지의 cost를 더한 것들의 최소를 구한다. 🡺 이와 같이 업데이트를 반복한다. (업데이트가 안될 때까지)  
  
  
  
  
  
  
 **Distance Vector Algorithm** Input: 각 노드의 이웃의 link cost  
 Output: 각 노드의 목적지를 가기위한 다음 hop과 corresponding cost  
 iterative (wait, recompute, notify), asynchronous(각 노드들이 가지고 있는 그림은 모두 달라서 결과가 다 다르게 나옴. // LS는 synchronous (각 node들은 모두 똑 같은 그림을 가지고 dijkstra를 돌리기 때문에 모두 똑 같은 결과가 나온다.  
 From time-to-time (때때로), 각 노드가 자신의 dv를 이웃에게 보낸다. 그 dv를 받은 이웃은 자신의 DV를 Bellman-ford equation: (**dx(y) = min { c(x,v) + dv(y) } for each node y ∈ N** )한 후 DV(routing info)를 업데이트한다. 만약 DV가 바뀌면 이웃한테 DV를 전해준다.  
 결국 DV는 좋은 소식은 빠르게, 안좋은 소식은 느리게 전달된다. (link cost가 안좋게 변하면, loop (count to infinity) 발생할 수 있음 🡪 poisoned reverse로 해결 (p.54)  
  
 Summary of LS & DV  
 Message complexity:   
 LS: n \* e (nodes \* links)  
 DV: n \* k (node’s \* number of neighbors)  
 Speed of convergence:  
 LS: O(n2)  
 DV: 느리다.