UNIT-3

Bellman-Ford Algorithm

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph. It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

This algorithm works correctly when some of the edges of the directed graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path between source and destination.

It is slower than Dijkstra's Algorithm but more versatile, as it capable of handling some of the negative weight edges.

Steps of Bellman-Ford Algorithm”

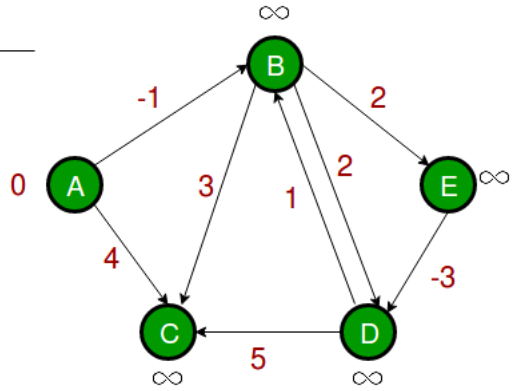
```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0


for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U
  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists
  return distance[], previous[]
```

Example
Let us understand the algorithm with following example graph.

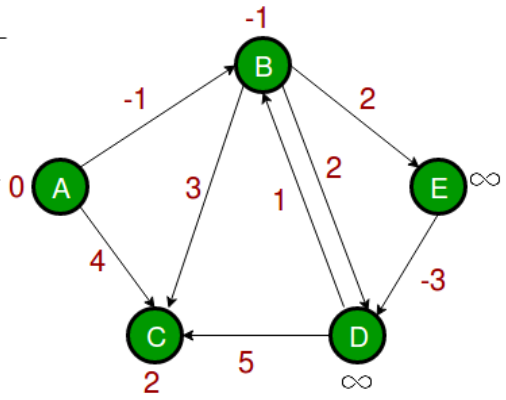
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times*.

A	B	C	D	E
0	∞	∞	∞	∞



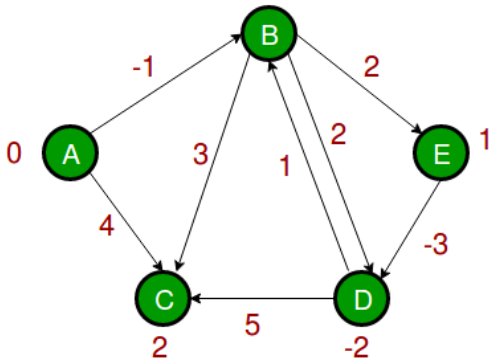
Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Distance-Vector Approach

A distance-vector routing (DVR) requires that a router inform its neighbours of topology changes periodically. Historically known as the old ARPANET routing algorithm.

- The Distance vector algorithm is iterative, asynchronous and distributed.
  - Distributed: It is distributed in that each node receives information from one or more of its directly attached neighbours, performs calculation and then distributes the result back to its neighbours.
  - Iterative: It is iterative in that its process continues until no more information is available to be exchanged between neighbours.
  - Asynchronous: It does not require that all of its nodes operate in the lock step with each other.
- The Distance vector algorithm is a dynamic algorithm.
- It is mainly used in ARPANET, and RIP.
- Each router maintains a distance table known as Vector.

Algorithm:

1. A router transmits its distance vector to each of its neighbors in a routing packet.
2. Each router receives and saves the most recently received distance vector from each of its neighbors.
3. A router recalculates its distance vector when:
   - It receives a distance vector from a neighbor containing different information than before.
   - It discovers that a link to a neighbor has gone down.

The DV calculation is based on minimizing the cost to each destination

Dx(y) = Estimate of least cost from x to y

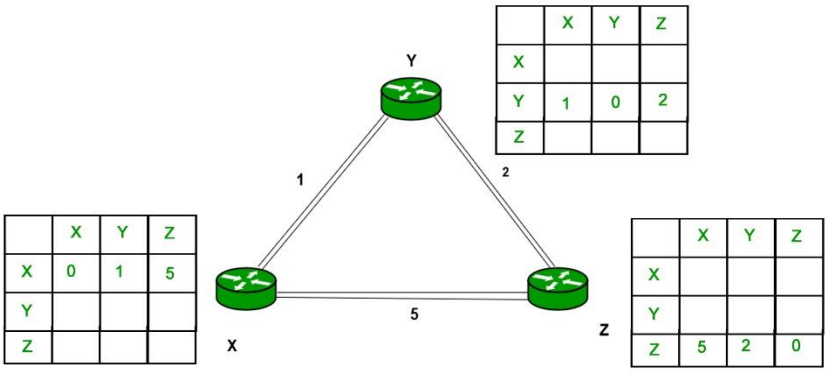
C(x,v) = Node x knows cost to each neighbor v

Dx = [Dx(y): y ∈ N] = Node x maintains distance vector

Node x also maintains its neighbors' distance vectors

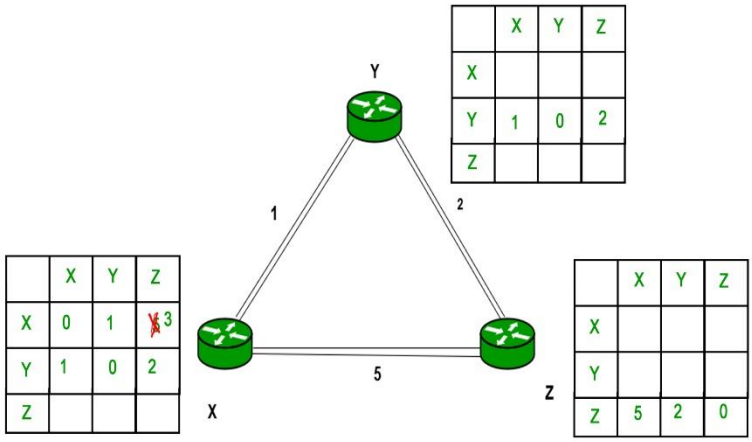
– For each neighbor v, x maintains Dv = [Dv(y): y ∈ N]

Example – Consider 3-routers X, Y and Z as shown in figure. Each router have their routing table. Every routing table will contain distance to the destination nodes.

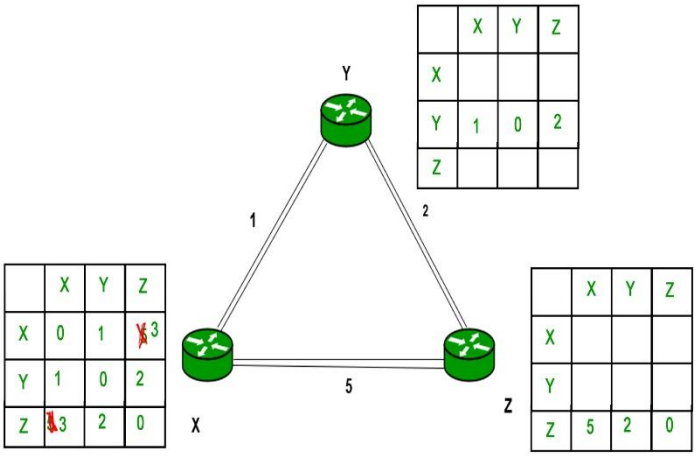


Consider router X , X will share it routing table to neighbors and neighbors will share it routing table to it to X and distance from node X to destination will be calculated using bellmen- ford equation.

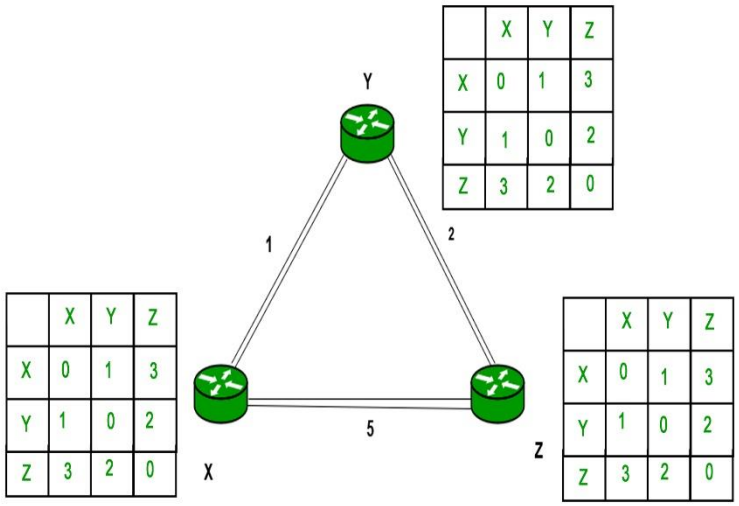
As we can see that distance will be less going from X to Z when Y is intermediate node(hop) so it will be update in routing table X.



Similarly for Z also –



Finally the routing table for all –



Advantages of Distance Vector routing:

- It is simpler to configure and maintain than link state routing.

Disadvantages of Distance Vector routing:

- It is slower to converge than link state.
- It is at risk from the count-to-infinity problem.
- It creates more traffic than link state since a hop count change must be propagated to all routers and processed on each router. Hop count updates take place on a periodic basis, even if there are no changes in the network topology, so bandwidth-wasting broadcasts still occur.

- For larger networks, distance vector routing results in larger routing tables than link state since each router must know about all other routers. This can also lead to congestion on WAN links.

Dijkstra Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

This problem is also called **single-source shortest paths problem.**

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Dijkstra algorithm pseudocode

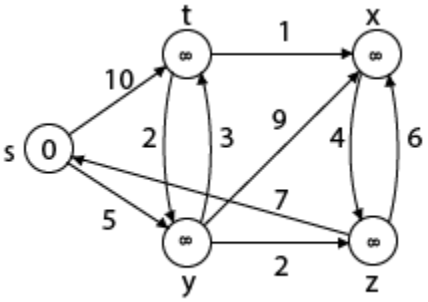
We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices. We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path. A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Example:



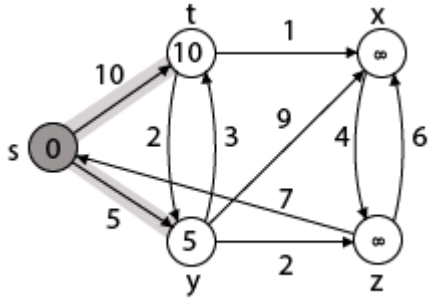
Step1: Q =[s, t, x, y, z]

We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices.

We make a stack, which contains those vertices which are selected after computation of shortest distance.

Firstly, we take 's' in stack M (which is a source)

Step 2: Now find the adjacent of s that are t and y.



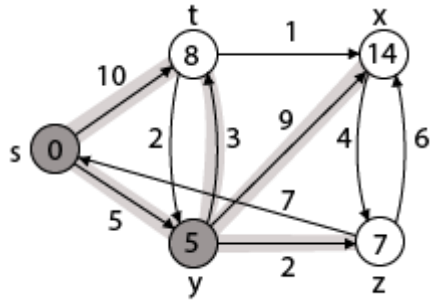
Case - (i) s → t
d [v] > d [u] + w [u, v]
d [t] > d [s] + w [s, t]
∞ > 0 + 10 [false condition]
Then **d [t] ← 10**
π [t] ← 5
Adj [s] ← t, y

Case - (ii) s→ y
d [v] > d [u] + w [u, v]
d [y] > d [s] + w [s, y]
∞ > 0 + 5 [false condition]
∞ > 5
Then **d [y] ← 5**
π [y] ← 5

By comparing case (i) and case (ii)
Adj [s] → t = 10, y = 5

y is shortest

y is assigned in 5 = [s, y]



Step 3: Now find the adjacent of y that is t, x, z.

Case - (i) y →t

d [v] > d [u] + w [u, v]

d [t] > d [y] + w [y, t]

10 > 5 + 3

10 > 8

Then d [t] ← 8

π [t] ← y

Case - (ii) y → x

d [v] > d [u] + w [u, v]

d [x] > d [y] + w [y, x]

∞ > 5 + 9

∞ > 14

Then d [x] ← 14

π [x] ← 14

Case - (iii) y → z

d [v] > d [u] + w [u, v]

d [z] > d [y] + w [y, z]

∞ > 5 + 2

∞ > 7

Then d [z] ← 7

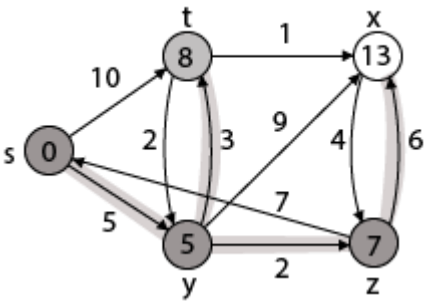
π [z] ← y

By comparing case (i), case (ii) and case (iii)

Adj [y] → x = 14, t = 8, z =7

z is shortest

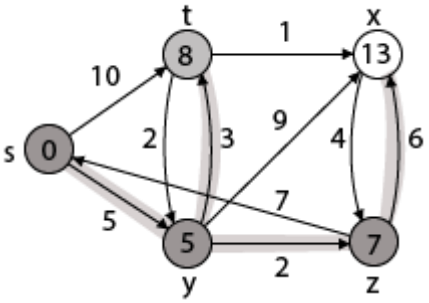
z is assigned in 7 = [s, z]



Step - 4 Now we will find adj [z] that are s, x

Case - (i) z → x
d [v] > d [u] + w [u, v]
d [x] > d [z] + w [z, x]
14 > 7 + 6
14 > 13
Then d [x] ← 13
π [x] ← z

Case - (ii) z → s
d [v] > d [u] + w [u, v]
d [s] > d [z] + w [z, s]
0 > 7 + 7
0 > 14
∴ This condition does not satisfy so it will be discarded.
Now we have x = 13.

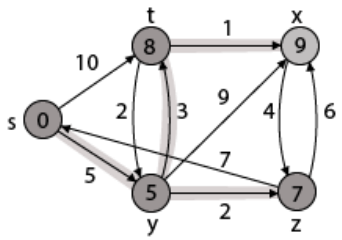


Step 5: Now we will find Adj [t]

Adj [t] → [x, y] [Here t is u and x and y are v]

Case - (i) t → x
d [v] > d [u] + w [u, v]
d [x] > d [t] + w [t, x]
13 > 8 + 1
13 > 9
Then d [x] ← 9
π [x] ← t

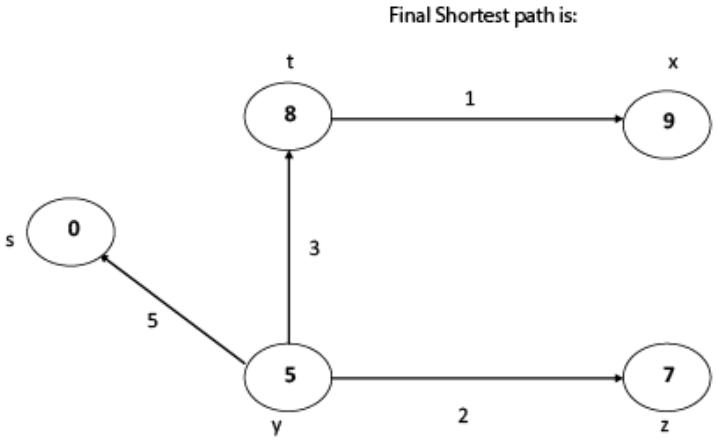
Case - (ii) t → y
d [v] > d [u] + w [u, v]
d [y] > d [t] + w [t, y]
5 > 10
∴ This condition does not satisfy so it will be discarded.



Thus we get all shortest path vertex as

Weight from s to y is 5
Weight from s to z is 7
Weight from s to t is 8
Weight from s to x is 9

These are the shortest distance from the source's' in the given graph.



Disadvantage of Dijkstra's Algorithm:

1. It does a blind search, so wastes a lot of time while processing.
2. It can't handle negative edges.
3. It leads to the acyclic graph and most often cannot obtain the right shortest path.
4. We need to keep track of vertices that have been visited.

Comparison between Bellman-Ford and Dijkstra Approach

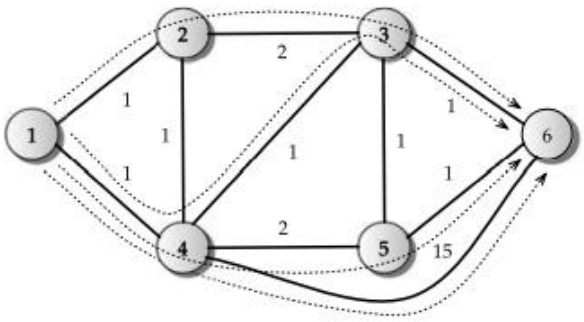
Bellman Ford's Algorithm	Dijkstra's Algorithm
Bellman Ford's Algorithm works when there is negative weight edge, it also detects the negative weight cycle.	Dijkstra's Algorithm doesn't work when there is negative weight edge.
The result contains the vertices which contains the information about the other vertices they are connected to.	The result contains the vertices containing whole information about the network, not only the vertices they are connected to.
It can easily be implemented in a distributed way.	It cannot be implemented easily in a distributed way.
It is more time consuming than Dijkstra's algorithm. Its time complexity is O(VE).	It is less time consuming. The time complexity is O(E logV).
Dynamic Programming approach is taken to implement the algorithm.	Greedy approach is taken to implement the algorithm.

Shortest Path Computation with Candidate Path Caching

There are certain networking environments where a list of possible paths is known or determined ahead of time; such a path list will be referred to as the candidate path list. Path caching refers to storing of a candidate path list at a node ahead of time. If through a distributed protocol mechanism the link cost is periodically updated, then the shortest path computation at a node becomes very simple when the candidate path list is already known.

Consider again the six-node network shown in Figure. Suppose that node 1 somehow knows that there are four paths available to node 6 as follows: 1-2-3-6, 1-4-3-6, 1-4-5-6, and 1-4-6; they are marked in Figure.

Using the link cost, we can then compute path cost for each path as shown in the table in Figure. Now, if we look for the least cost path, we will find that path 1-4-3-6 is the most preferred path due to its lowest end-to-end cost. Suppose now that in the next time period, the link cost for link 4-3 changes from 1 to 5. If we know the list of candidate paths, we can then recompute the path cost and find that path 1-4-3-6 is no longer the least cost; instead, both 1-2-3-6 and 1-4-5-6 are now the shortest paths—either can be chosen based on a tie-breaker rule.



Path	Cost
1-2-3-6	$d_{12} + d_{23} + d_{36} = 4$
1-4-3-6	$d_{14} + d_{43} + d_{36} = 3$
1-4-5-6	$d_{14} + d_{45} + d_{56} = 4$
1-4-6	$d_{14} + d_{46} = 16$

We will now write the shortest path calculation in the presence of path caching in a generic way, considering that this calculation is done at time t. We consider a candidate path p between nodes i and node j, and its cost at time t as

$$\widehat{D}_{ij/p}(t) = \sum_{\text{link } l\text{-}m \text{ in path } p} d^{i}_{lm}(t),$$

where $di_{lm}(t)$ is the cost of link l-m at time t as known to node i, and the summation is over all such links that are part of path p. The list of candidate paths for the pair i and j will be denoted by Pij; the best path will be identified by p. The procedure to compute the shortest path is given in Algorithm

Algorithm:

At source node i, a list of candidate paths $\mathcal{P}_{ij}$ to destination node j is available, and link cost, $d^i_{lm}(t)$, of link l-m at time t is known:

// Initialize the least cost:

$\widehat{D}_{ij}(t) = \infty$

// Consider each candidate path in the list

for (p in $\mathcal{P}_{ij}$) do

  $\widehat{D}_{ij/p}(t) = 0$

  for (link l-m in path p) do // add up cost of links for this path

    $\widehat{D}_{ij/p}(t) = \widehat{D}_{ij/p}(t) + d^i_{lm}(t)$

  end for

  if ($\widehat{D}_{ij/p}(t) < \widehat{D}_{ij}(t)$) then // if this is cheaper, note it

    $\widehat{D}_{ij}(t) = \widehat{D}_{ij/p}(t)$

    $\widehat{p} = p$

  end if

end do

It is important to note that the candidate path list is not required to include all possible paths between node i and j, only a sublist of paths that are, for some reason, preferable to consider for a particular networking environment.

In a communication network, this approach of computing the shortest path involves a trade-off between storage and time complexity. That is, by storing multiple candidate paths ahead of time, the actual computation is simple when new link costs are received. The set of candidate paths can be determined using, for example, the K-shortest path algorithm; since the interest in the case of path caching is obtain a good working set, any reasonable link cost can be assumed; for example, we can set all link costs to 1 (known also as hop count) and use Algorithm to obtain a set of K candidate paths.

Widest Path Computation with Candidate Path Caching

So far, we have assumed that the shortest path is determined based on the additive cost property. There are many networking environments in which the additive cost property is not applicable; for example, dynamic call routing in the voice telephone network and quality of service-based routing. Thus, determining paths when the cost is nonadditive is also an important problem in network routing; an important class among the nonadditive cost properties is concave cost property that leads to widest path routing. We will first start with the case in which path caching is used, so that it is easy to transition and compare where and how the nonadditive concave case is different from the additive case described in the previous section.

Suppose a network link has a certain bandwidth available, sometimes referred to as residual capacity; to avoid any confusion, we will denote the available bandwidth by b_{lm} for link l-m, as opposed to d_{lm} for the additive case. Note that $b_{lm} = 0$ then means that the link is not feasible since there is no bandwidth; we can also set $b_{lm} = 0$ if there is no link between nodes l and m (compare this with $d_{lm} = \infty$ for the additive case). We start with a simple illustration. Consider a path between node 1 and node 2 consisting of three links: the first link has 10 units of bandwidth available; the second link has 5 units of bandwidth available, and the third link has 7 units of bandwidth available. Now, if we say the cost of this path is additive, i.e., 22(= 10 + 5 + 7), it is unlikely to make any sense.

There is another way to think about it. Suppose that we have new requests coming in, each requiring a unit of dedicated bandwidth for a certain duration. What is the maximum number of requests this path can handle? It is easy to see that this path would be able to handle a maximum of five additional requests simultaneously since if it were more than five, the link in the middle in this case would not be able to handle more than five requests. That is, we arrive at the availability of the path by doing min{10, 5, 7} = 5. Thus, the path "cost" is 5; certainly, this is a strange definition of a path cost; it is easier to see this as the width of a path, for all links l-m that make up a path p, we can write the width of the path as

$$\widehat{B}_{ij/p}(t) = \min_{\text{link } l\text{-}m \text{ in path } p} \left\{ b^i_{lm}(t) \right\}.$$

Regardless, the important point to note is that this path cost is computed using a nonadditive cost property, in this case the minimum function. It may be noted that the minimum function is not the only nonadditive cost property possible for defining cost of a path; there are certainly other possible measures, such as the nonadditive multiplicative property. Now consider a list of candidate paths; how do we define the most preferable path? One way to define it is to find the path with the largest amount of available bandwidth. This is actually easy to do once the path "cost" for each path is determined since we can then take the maximum of all such paths. Consider the topology shown in Figure 2.7 with available bandwidth on each link as marked. Now consider three possible paths between node 1 and node 5:

Path	Cost
1-2-3-5	$\min\{b_{12}, b_{23}, b_{35}\} = 10$
1-4-3-5	$\min\{b_{14}, b_{43}, b_{35}\} = 15$
1-4-5	$\min\{b_{14}, b_{45}\} = 20$

Algorithm:

At source node i, a list of candidate paths $\mathcal{P}_{ij}$ to destination node j is available, and link bandwidth, $b^i_{lm}(t)$, of link l-m at time t is known:
// Initialize the least bandwidth:
$\widehat{B}_{ij}(t) = 0$
for p in $\mathcal{P}_{ij}$ do
  $\widehat{B}_{ij/p}(t) = \infty$
  for (link l-m in path p) do // find bandwidth of the bottleneck link
  $\widehat{B}_{ij/p}(t) = \min\left\{\widehat{B}_{ij/p}(t), b^i_{lm}(t)\right\}$
  end for
  if $(\widehat{B}_{ij/p}(t) \succ \widehat{B}_{ij}(t))$ then // if this has more bandwidth, note it
    $\widehat{B}_{ij}(t) = \widehat{B}_{ij/p}(t)$
    $\widehat{p} = p$
  end if
end do

It is easy to see that the third path, 1-4-5, has the most bandwidth and is thus the preferred path. This means that we need to do a maximum over all paths in the case of the nonadditive property to find the

widest path as opposed to the minimum over all paths when additive cost property is used. A widest path so selected is sometimes referred to as the maximal residual capacity path. The procedure is presented in detail. It is helpful to contrast this algorithm with its counterpart, previous algorithm, where the additive cost property was used.

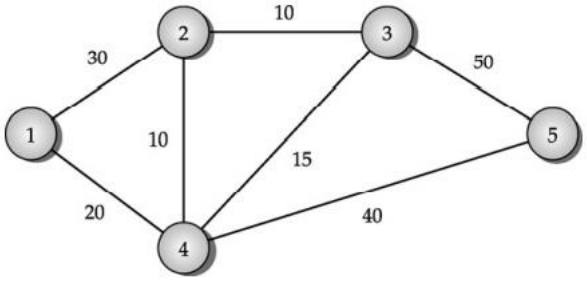


FIGURE 2.7 Network example for widest path routing.

Widest Path Algorithm

We are coming back full circle to no path caching for widest path routing algorithms. We present two approaches: first we show an extension of Dijkstra's shortest path first algorithm; next, we extend the Bellman–Ford algorithm.

Dijkstra-Based Approach

When there is no path caching, the algorithm is very similar to Dijkstra's algorithm.

Consider the network topology shown in above Figure 2.7

Algorithm:

1. Discover list of nodes in the network, $\mathcal{N}$ and available bandwidth of link k-m, $b^i_{km}(t)$, as known to node i at the time of computation, t.
2. Initially, consider only source node i in the set of nodes considered, i.e., $\mathcal{S} = \{i\}$; mark the set with all the rest of the nodes as $\mathcal{S}'$. Initialize

   $$\underline{B}_{ij}(t) = b^i_{ij}(t).$$

3. Identify a neighboring node (intermediary) k not in the current list $\mathcal{S}$ with the maximum bandwidth from node i, i.e., find $k \in \mathcal{S}'$ such that $\underline{B}_{ik}(t) = \max_{m \in \mathcal{S}'} \underline{B}_{im}(t)$

   Add k to the list $\mathcal{S}$, i.e., $\mathcal{S} = \mathcal{S} \cup \{k\}$

   Drop k from $\mathcal{S}'$, i.e., $\mathcal{S}' = \mathcal{S}' \backslash \{k\}$.

   If $\mathcal{S}'$ is empty, stop.
4. Consider nodes in $\mathcal{S}'$ to update maximum bandwidth path, i.e., for $j \in \mathcal{S}'$

   $$\underline{B}_{ij}(t) = \max\{\underline{B}_{ij}(t), \min\{\underline{B}_{ik}, b^i_{kj}(t)\}\}. \qquad (2.7.1)$$

   Go to Step 3.

Iterative steps based on above algorithm:

Iteration	List, $\mathcal{S}$	$\underline{B}_{12}$	Path	$\underline{B}_{13}$	Path	$\underline{B}_{14}$	Path	$\underline{B}_{15}$	Path
1	{1}	30	1-2	0	–	20	1-4	0	–
2	{1, 2}	30	1-2	10	1-2-3	20	1-4	0	–
3	{1, 2, 4}	30	1-2	15	1-4-3	20	1-4	20	1-4-5
4	{1, 2, 4, 3}	30	1-2	15	1-4-3	20	1-4	20	1-4-5
5	{1, 2, 4, 3, 5}	30	1-2	15	1-4-3	20	1-4	20	1-4-5

the distributed time-dependent case from the point of view of node 1, i.e., suppose that node 1 wants to find the path with most available bandwidth to all other nodes in the network. Then, initially, S = {1} and S = {2, 3, 4, 5}, and the widest paths to all nodes that are direct neighbors of node 1 can be readily found while for the rest, the "cost" remains at 0, i.e.,

$$B12 = 30,\ B14 = 20,\ B13 = B15 = 0.$$

Since $\max_{j \in S} B1j = 30$ is attained for j = 2, we add node 2 to list S. Thus, we have the updated lists: S = {1, 2} and S' = {3, 4, 5}.

Now for j not in S, we update the available bandwidth to see if it is better than going via node 2, as follows:

$$B13 = \max\{B13, \min\{B12,b23\}\} = \max\{0, \min\{30, 10\}\} = 10 \text{ // use 1-2-3}$$

$$B14 = \max\{B14, \min\{B12,b24\}\} = \max\{20, \min\{30, 10\}\} = 20 \text{ // stay on 1-2}$$

$$B15 = \max\{B15, \min\{B12,b25\}\} = \max\{0, \min\{30, 0\}\} = 0 \text{ // no change}$$

Now we are in the second pass of the algorithm. This time, $\max_{j \in S} B1j = \max\{B13, B14, B15\} = 20$. This is attained for j = 4. Thus, S becomes {1, 2, 4}. Updating the available bandwidth to check via node 4 will be as follows:

$$B13 = \max\{B13, \min\{B14,b43\}\} = \max\{10, \min\{20, 15\}\} = 15 \text{ // use 1-4-3}$$

$$B15 = \max\{B15, \min\{B14,b45\}\} = \max\{0, \min\{20, 40\}\} = 20 \text{ // use 1-4-5}$$

This time, j = 3 will be included in S. Thus, S becomes {1, 2, 4, 3}. There is no further improvement in the final pass of the algorithm.

Bellman–Ford-Based Approach

The widest path algorithm that uses the Bellman–Ford-based approach is strikingly similar to the Bellman–Ford shortest path routing algorithm.

Initialize

$$\overline{B}_{ii}(t) = 0; \quad \overline{B}_{ij}(t) = 0, \quad \text{(for node } j \text{ that node } i \text{ is aware of).}$$

For (nodes j that node i is aware of) do

$$\overline{B}_{ij}(t) = \max_{k \text{ directly connected to } i} \min\left\{b_{ik}(t), \overline{B}^{i}_{kj}(t)\right\}, \quad \text{for } j \neq i.$$

k-Shortest Paths Algorithm

We now go back to the class of shortest path algorithms to consider an additional case. In many networking situations, it is desirable to determine the second shortest path, the third shortest path, and so on, up to the k-th shortest path between a source and a destination. Algorithms used for determining paths beyond just the shortest paths are generally referred to as k-shortest paths algorithms.

A simple way to generate additional paths would be to start with, say Dijkstra's shortest path first algorithm, to determine the shortest path; then, by temporarily deleting each link on the shortest path one at a time, we can consider the reduced graph where we can apply again

1. Initialize $k := 1$.
2. Find the shortest path $\mathcal{P}$ between source (i) and destination (j) in graph $\mathcal{G}$, using Dijkstra's Algorithm.

   Add $\mathcal{P}$ to permanent list $\mathcal{K}$, i.e., $\mathcal{K} := \{\mathcal{P}\}$.

   If $K = 1$, stop.

   Add $\mathcal{P}$ to set $\mathcal{X}$ and pair $(\mathcal{P}, i)$ to set $\mathcal{S}$, i.e., $\mathcal{X} := \{\mathcal{P}\}$ and $\mathcal{S} := \{(\mathcal{P}, i)\}$.
3. Remove $\mathcal{P}$ from $\mathcal{X}$, i.e., $\mathcal{X} := \mathcal{X} \backslash \{\mathcal{P}\}$.
4. Find the unique pair $(\mathcal{P}, w) \in \mathcal{S}$, and corresponding deviation node w associated with $\mathcal{P}$.
5. For each node v, except j, on subpath of $\mathcal{P}$ from w to j ($sub_{\mathcal{P}}(w, j)$):

   Construct graph $\mathcal{G}'$ by removing the following from graph $\mathcal{G}$:

   (a) All the vertices on subpath of $\mathcal{P}$ from i to v, except v.

   (b) All the links incident on these deleted vertices.

   (c) Links outgoing from v toward j for each $\mathcal{P}' \in \mathcal{K} \cup \{\mathcal{P}\}$, such that $sub_{\mathcal{P}}(i, v) = sub_{\mathcal{P}'}(i, v)$.

   Find the shortest path $\mathcal{Q}'$ from v to j in graph $\mathcal{G}'$ using Dijkstra's Algorithm.

   Concatenate subpath of $\mathcal{P}$ from i to v and path $\mathcal{Q}'$, i.e., $\mathcal{Q} = sub_{\mathcal{P}}(i, v) \oplus \mathcal{Q}'$.

   Add $\mathcal{Q}$ to $\mathcal{X}$ and pair $(\mathcal{Q}, v)$ to $\mathcal{S}$, i.e., $\mathcal{X} := \mathcal{X} \cup \{\mathcal{Q}\}$ and $\mathcal{S} := \mathcal{S} \cup \{(\mathcal{Q}, v)\}$.
6. Find the shortest path $\mathcal{P}$ among the paths in $\mathcal{X}$ and add $\mathcal{P}$ to $\mathcal{K}$, i.e., $\mathcal{K} := \mathcal{K} \cup \mathcal{P}$.
7. Increment k by 1.
8. If $k < K$ and $\mathcal{X}$ is not empty, go to Step 4, else stop.

Dijkstra's shortest path first algorithm. This will then give us paths that are longer than the shortest path. By identifying the cost of each of these paths, we can sort them in order of successively longer paths. For example, consider finding k-shortest paths from node 1 to node 6. Here, the shortest path is 1-4-3-6 with path cost 3. Through this procedure, we can find longer paths such as 1-2-3-6 (path cost 4), 1-4-5-6 (path cost 4), and 1-4-3-5-6 (path cost 4). It is easy to see that paths so determined may have one or more links in common.

Suppose that we want to find k-shortest link disjoint paths. In this case, we need to temporarily delete all the links on the shortest path and run Dijkstra's algorithm again on the reduced graph—this will then give the next shortest link disjoint path; we can continue this process until we find k-shortest link disjoint paths. Sometimes it might not be possible to find two or more link disjoint paths, if the reduced graph is isolated into more than one network. Consider again Figure. Here, the shortest path from node 1 to node 6 is 1-4-3-6 with path cost 3. If we temporarily delete the links in this path, we find the next link-disjoint shortest path to be 1-2-4-5-6 of path cost 5. If we now delete links in this path, node 1 becomes isolated in the newly obtained reduced graph.

In Algorithm, we present a k-shortest path algorithm that is based on an idea, for additional references for this method. In this algorithm, a fairly complicated process is applied beyond finding the shortest path. For example, it uses an auxiliary list S in order to track/determine longer paths. This is meant for die-hard readers, though. A description with each step is included in Algorithm to convey the basic idea behind this algorithm.

Finally, recall that we discussed widest path computations with candidate path caching; such candidate paths to be cached can also be determined using a k-shortest paths algorithm. Typically, in such situations, the link cost for all links can be set to 1 because usually hop-length-based k-shortest paths are sufficient to determine candidate paths to cache.

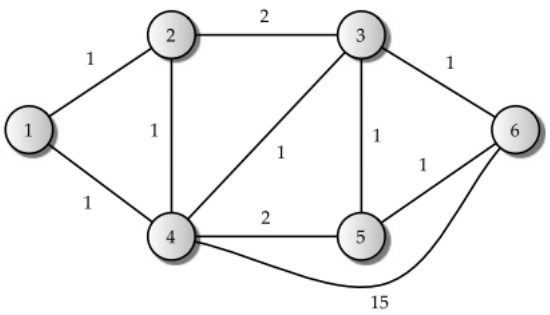


Figure 2.1