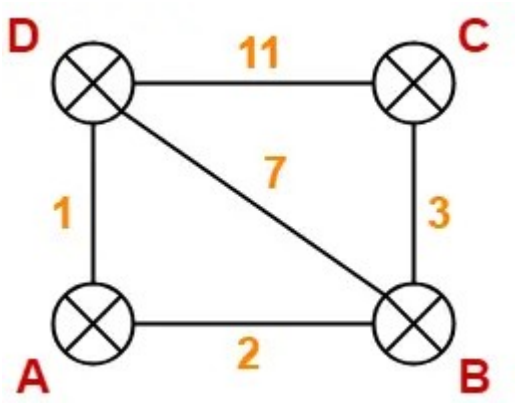
Distance Vector Routing Algorithm

A distance-vector routing protocol requires that a router informs its neighbors of topology changes periodically and, in some cases, when a change is detected in the topology of a network. Compared to link-state protocols, which require a router to inform all the nodes in a network of topology changes, distance-vector routing protocols have less computational complexity and message overhead.

Distance Vector means that Routers are advertised as vector of distance and direction. 'Direction' is represented by next hop address and exit interface, whereas 'Distance' uses metrics such as hop count.



Step-01:

Each router prepares its routing table using its local knowledge.

Routing table prepared by each router is shown below- At Router A-

Destination	Distance	Next Hop
A	0	A
B	2	B
C	∞	–
D	1	D

At Router B-

Destination	Distance	Next Hop
A	2	A
B	0	B
C	3	C
D	7	D

At Router C-

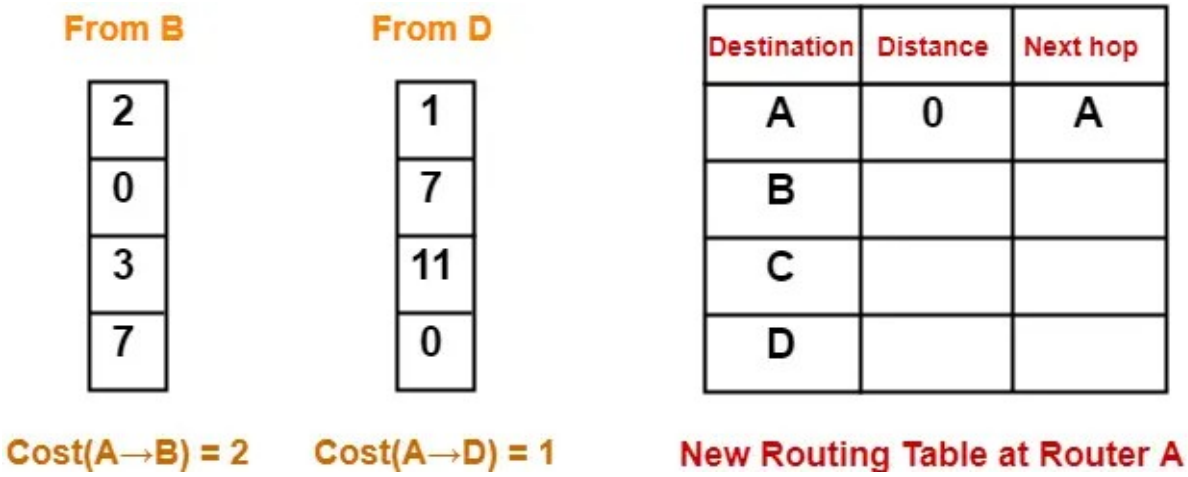
Destination	Distance	Next Hop
A	∞	–
B	3	B
C	0	C
D	11	D

At Router D-

Destination	Distance	Next Hop
A	1	A
B	7	B
C	11	C
D	0	D

At Router A-

- Router A receives distance vectors from its neighbors B and D.
- Router A prepares a new routing table as-



- Cost of reaching destination B from router A = min { 2+0 , 1+7 } = 2 via B.
- Cost of reaching destination C from router A = min { 2+3 , 1+11 } = 5 via B.
- Cost of reaching destination D from router A = min { 2+7 , 1+0 } = 1 via D.

Explanation For Destination B

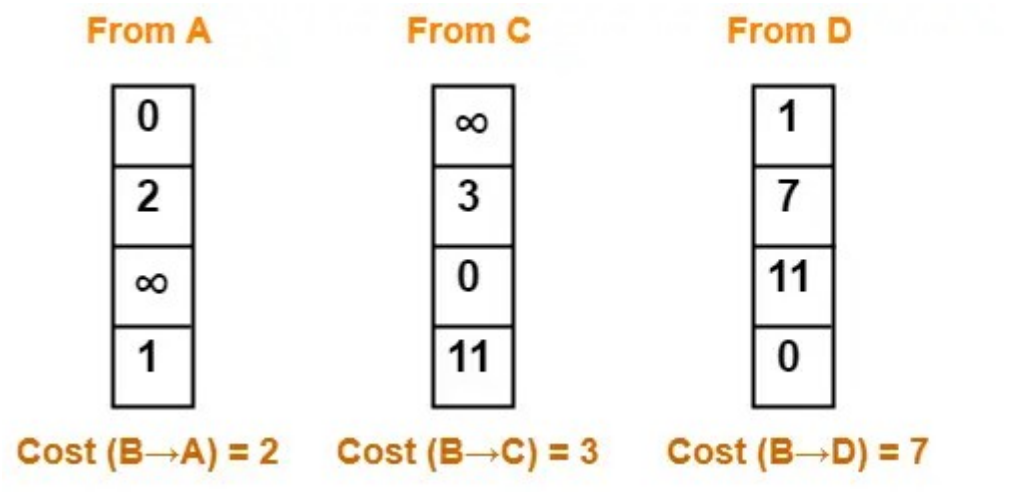
- Router A can reach the destination router B via its neighbor B or neighbor D.
- It chooses the path which gives the minimum cost.
- Cost of reaching router B from router A via neighbor B = Cost (A→B) + Cost (B→B)= **2 + 0** = 2
- Cost of reaching router B from router A via neighbor D = Cost (A→D) + Cost (D→B) = **1 + 7** = 8
- Since the cost is minimum via neighbor B, so router A chooses the path via B.
- It creates an entry (2, B) for destination B in its new routing table.
- Similarly, we calculate the shortest path distance to each destination router at every router.

Thus, the new routing table at router A is-

Destination	Distance	Next Hop
A	0	A
B	2	B
C	5	B
D	1	D

At Router B-

- Router B receives distance vectors from its neighbors A, C and D.
- Router B prepares a new routing table as-
-



• Cost of reaching destination A from router B = min { 2+0 , 3+∞ , 7+1 } = 2 via A.

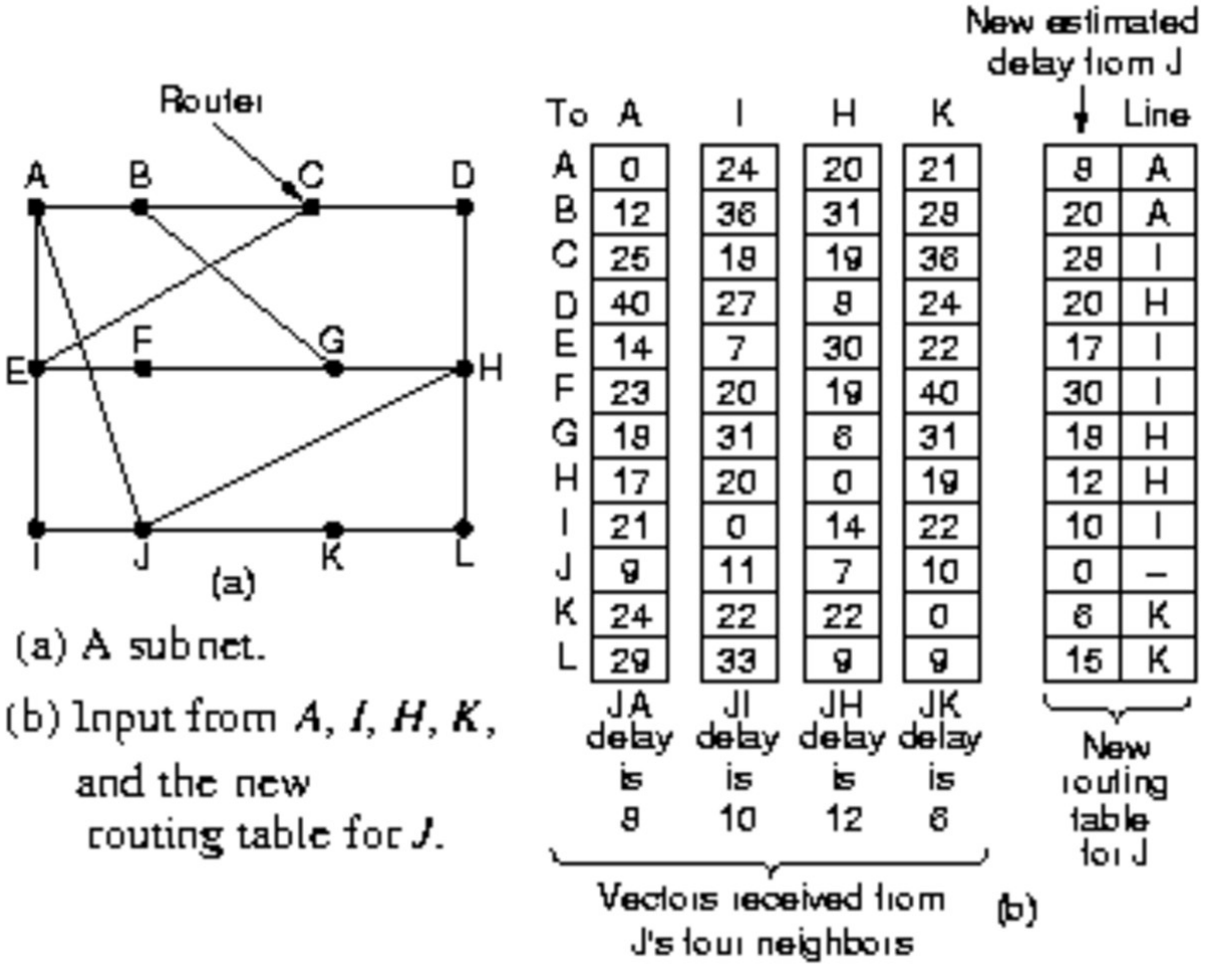
Cost of reaching destination C from router Thus, the new routing table at router B is-

Destination	Distance	Next Hop
A	2	A
B	0	B
C	3	C
D	3	A

At Router C-

• = min { 2+∞ , 3+0 , 7+11 } = 3 via C.

• Cost of reaching destination D from router B = min { 2+1 , 3+11 , 7+0 } = 3 via A.



Distance vector algorithm routing implementation in C

```
#include<stdio.h>
struct node{
unsigned dist[20];
unsigned from[20];
}rt[10];

int main()
{
int dmat[20][20];
```

```
int n,i,j,k,count=0;
 printf("\nEnter the number of nodes : ");
scanf("%d",&n);
printf("\nEnter the cost matrix :\n");
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
{
 scanf("%d",&dmat[i][j]);
dmat[i][i]=0;
 rt[i].dist[j]=dmat[i][j];
rt[i].from[j]=j; }
 do {
count=0;
for(i=0;i<n;i++)
for(j=0;j<n;j++)
for(k=0;k<n;k++)
 if(rt[i].dist[j]>dmat[i][k]+rt[k].dist[j])
{ rt[i].dist[j]=rt[i].dist[k]+rt[k].dist[j]; rt[i].from[j]=k; count+
+; } }while(count!=0);
for(i=0;i<n;i++)
 {
 printf("\n\nState value for router %d is \n",i+1);
for(j=0;j<n;j++)
{
 printf("\t\nnode %d via %dDistance%d",j+1,rt[i].from[j]
+1,rt[i].dist[j]); }
}
 printf("\n\n");
}
```

output

Enter the number of nodes : 4

Enter the cost matrix :
0 3 5 99
3 0 99 1
5 4 0 2
99 1 2 0

State value for router 1 is

node 1 via 1 Distance0
node 2 via 2 Distance3
node 3 via 3 Distance5
node 4 via 2 Distance4

State value for router 2 is

node 1 via 1 Distance3
node 2 via 2 Distance0
node 3 via 4 Distance3
node 4 via 4 Distance1

State value for router 3 is

node 1 via 1 Distance5
node 2 via 4 Distance3
node 3 via 3 Distance0
node 4 via 4 Distance2

State value for router 4 is

node 1 via 2 Distance4
node 2 via 2 Distance1
node 3 via 3 Distance2
node 4 via 4 Distance0

_

What is socket programming?

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.

Stages for server

1. Socket creation:

int sockfd = socket(domain, type, protocol)

- sockfd: socket descriptor, an integer (like a file-handle)
- domain: integer, specifies communication domain. We use AF_LOCAL as defined in the POSIX standard for communication between processes on the same host. For communicating between processes on different hosts connected by IPV4, we use AF_INET and AF_I NET 6 for processes connected by IPV6.
- type: communication type
  SOCK_STREAM: TCP(reliable, connection oriented)
  SOCK_DGRAM: UDP(unreliable, connectionless)
- protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)

2. Setsockopt:

This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

*int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);*

3. Bind:

*int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);*

After the creation of the socket, the bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR_ANY to specify the IP address.

4. Listen:

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

5. Accept:

*int new_socket= accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);*

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, the connection is established between client and server, and they are ready to transfer data.

Stages for Client

- Socket connection: Exactly same as that of server's socket creation
- Connect: The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

  *int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);*

Implementation
Here we are exchanging one hello message between server and client to demonstrate the client/server model.

Server.c

- C

```
// Server side C/C++ program to demonstrate Socket
// programming
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <sys/socket.h>
#include <unistd.h>
#define PORT 8080
int main(int argc, char const* argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[1024] = { 0 };
    char* hello = "Hello from server";

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port 8080
    if (setsockopt(server_fd, SOL_SOCKET,
                   SO_REUSEADDR | SO_REUSEPORT, &opt,
                   sizeof(opt))) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Forcefully attaching socket to the port 8080
    if (bind(server_fd, (struct sockaddr*)&address,
             sizeof(address))
        < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    if ((new_socket
         = accept(server_fd, (struct sockaddr*)&address,
                  (socklen_t*)&addrlen))
        < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }
```

```
    valread = read(new_socket, buffer, 1024);
    printf("%s\n", buffer);
    send(new_socket, hello, strlen(hello), 0);
    printf("Hello message sent\n");

    // closing the connected socket
    close(new_socket);
    // closing the listening socket
    shutdown(server_fd, SHUT_RDWR);
    return 0;
}
```

client.c

- C

```
// Client side C/C++ program to demonstrate Socket
// programming
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 8080

int main(int argc, char const* argv[])
{
    int status, valread, client_fd;
    struct sockaddr_in serv_addr;
    char* hello = "Hello from client";
    char buffer[1024] = { 0 };
    if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary
    // form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)
        <= 0) {
        printf(
            "\nInvalid address/ Address not supported \n");
        return -1;
    }
```

```
    if ((status
         = connect(client_fd, (struct sockaddr*)&serv_addr,
                   sizeof(serv_addr)))
        < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
    send(client_fd, hello, strlen(hello), 0);
    printf("Hello message sent\n");
    valread = read(client_fd, buffer, 1024);
    printf("%s\n", buffer);

    // closing the connected socket
    close(client_fd);
    return 0;
}
```

Compiling:

```
gcc client.c -o client
```

```
gcc server.c -o server
```

Output:

```
Client:Hello message sent
```

```
Hello from server
```

```
Server:Hello from client
```

```
Hello message sent
```