**Lab Practical #11:**

To develop network using distance vector routing protocol and link state routing protocol.

**Practical Assignment #11:**

1. **C Program: Distance Vector Routing Algorithm using Bellman Ford's Algorithm.**
2. #include<stdio.h>
3. struct node
4. {
5. unsigned dist[20];
6. unsigned from[20];
7. }rt[10];
8. int main()
9. {
10. int dmat[20][20];
11. int n,i,j,k,count=0;
12. printf("\nEnter the number of nodes : ");
13. scanf("%d",&n);
14. printf("\nEnter the cost matrix :\n");
15. for(i=0;i<n;i++)
16. for(j=0;j<n;j++)
17. {
18. scanf("%d",&dmat[i][j]);
19. dmat[i][i]=0;
20. rt[i].dist[j]=dmat[i][j];
21. rt[i].from[j]=j;
22. }
23. do
24. {
25. count=0;
26. for(i=0;i<n;i++)
27. for(j=0;j<n;j++)
28. for(k=0;k<n;k++)
29. if(rt[i].dist[j]>dmat[i][k]+rt[k].dist[j])
30. {
31. rt[i].dist[j]=rt[i].dist[k]+rt[k].dist[j];
32. rt[i].from[j]=k;
33. count++;
34. }
35. }while(count!=0);
36. for(i=0;i<n;i++)
37. {
38. printf("\n\nState value for router %d is \n",i+1);
39. for(j=0;j<n;j++)
40. {
41. printf("\t\nnode %d via %d Distance%d",j+1,rt[i].from[j]+1,rt[i].dist[j]);
42. }
43. }
44. printf("\n\n");
45. }

**2. C Program: Link state routing algorithm.**

#include "global.h"

#include <assert.h>

#include <limits.h>

#include <stdlib.h>

#include <stdio.h>

#include <arpa/inet.h>

#include <string.h>

#define INFINITY                INT\_MAX

#define UNDEFINED               (-1)

#define INDEX(x, y, nnodes)     ((x) + (nnodes) \* (y))

struct node\_list {

    char \*\*nodes;

    int nnodes;

    int unsorted;

};

int nl\_index(struct node\_list \*nl, char \*node);

struct node\_list \*nl\_create(void) {

    return (struct node\_list \*) calloc(1, sizeof(struct node\_list));

}

int nl\_nsites(struct node\_list \*nl){

    return nl->nnodes;

}

void nl\_add(struct node\_list \*nl, char \*node){

    /\* No duplicate nodes.

     \*/

    if (nl\_index(nl, node) != -1) {

        return;

    }

    /\* Create a copy of the site.

     \*/

    int len = strlen(node);

    char \*copy = malloc(len + 1);

    strcpy(copy, node);

    /\* Add this copy to the list.

     \*/

    nl->nodes= (char \*\*) realloc(nl->nodes, sizeof(char \*) \* (nl->nnodes + 1));

    nl->nodes[nl->nnodes++] = copy;

    nl->unsorted = 1;

}

int nl\_compare(const void \*e1, const void \*e2){

    const char \*\*p1 = (const char \*\*) e1, \*\*p2 = (const char \*\*) e2;

    return strcmp(\*p1, \*p2);

}

void nl\_sort(struct node\_list \*nl){

    qsort(nl->nodes, nl->nnodes, sizeof(char \*), nl\_compare);

    nl->unsorted = 0;

}

/\* Return the rank of the given site in the given site list.

 \*/

int nl\_index(struct node\_list \*nl, char \*node){

    /\* Sort the list if not yet sorted.

     \*/

    if (nl->unsorted) {

        nl\_sort(nl);

    }

    /\* Binary search.

     \*/

    int lb = 0, ub = nl->nnodes;

    while (lb < ub) {

        int i = (lb + ub) / 2;

        int cmp = strcmp(node, nl->nodes[i]);

        if (cmp < 0) {

            ub = i;

        }

        else if (cmp > 0) {

            lb = i + 1;

        }

        else {

            return i;

        }

    }

    return -1;

}

char \*nl\_name(struct node\_list \*nl, int index){

    if (index < 0) {

        return "UNDEFINED";

    }

    return nl->nodes[index];

}

void nl\_destroy(struct node\_list \*nl){

    int i;

    for (i = 0; i < nl->nnodes; i++) {

        free(nl->nodes[i]);

    }

    free(nl->nodes);

    free(nl);

}

/\* Set the distance from src to dst.

 \*/

void set\_dist(struct node\_list \*nl, int graph[], int nnodes, char \*src, char \*dst, int dist){

    int x = nl\_index(nl, src), y = nl\_index(nl, dst);

    if (x < 0 || y < 0) {

        fprintf(stderr, "set\_dist: bad source or destination\n");

        return;

    }

    graph[INDEX(x, y, nnodes)] = dist;

   // graph[INDEX(y, x, nnodes)] = dist;

}

char\* addr\_to\_string (struct sockaddr\_in addr) {

    char\* addr\_string = malloc(40);

    strcpy(addr\_string, inet\_ntoa(addr.sin\_addr));

    strcat(addr\_string, ":");

    char\* port = malloc(12);

    sprintf(port, "%d", ntohs(addr.sin\_port));

    strcat(addr\_string, port);

    free(port);

    return addr\_string;

}

struct sockaddr\_in string\_to\_addr(char\* string) {

    char \*port = index(string, ':');

    \*port++ = '\0';

    struct sockaddr\_in addr;

    memset((void\*)&addr, 0, sizeof(addr));

    addr\_get(&addr, string, atoi(port));

    \*--port = ':';

    return addr;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    Dijkstra's algorithm

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Dijkstra's algorith.  graph[INDEX(x, y, nnodes)] contains the

 \* distance of node x to node y.  nnodes is the number of nodes.  src

 \* is that starting node.  Output dist[x] gives the distance from src

 \* to x.  Output prev[x] gives the last hop from src to x.

 \*/

void dijkstra(int graph[], int nnodes, int src, int dist[], int prev[]){

      int \*visited = malloc(sizeof(int) \* nnodes); // mark whether the node is visited

      int count, mindistance, nextnode, i, j;

      for (i = 0; i < nnodes; i++) {

            visited[i] = 0;

            if (graph[INDEX(src, i, nnodes)] == 1 || graph[INDEX(src, i, nnodes)] == 0) {

                dist[i] = graph[INDEX(src, i, nnodes)];

                prev[i] = src;

            } else {

                dist[i] = INFINITY;

            }

      }

      dist[src] = 0;

      visited[src] = 1;

      prev[src] = UNDEFINED;    // src has no prev

      for(count = 0; count < nnodes; count++) {

          mindistance = INFINITY;

          for (i = 0; i < nnodes; i++) {

            if(dist[i] < mindistance && !visited[i]) {

                mindistance = dist[i];

                nextnode = i;

            }

          }

          visited[nextnode] = 1;

          for (j = 0; j < nnodes; j++) {

               if(!visited[j] && (graph[INDEX(nextnode, j, nnodes)] != INFINITY)

                  && dist[nextnode] + graph[INDEX(nextnode, j, nnodes)] < dist[j]){

                  dist[j] = dist[nextnode] + graph[INDEX(nextnode, j, nnodes)];

                  prev[j] = nextnode;

               }

           }

    }

}