Application of Graph
**Application of Graph : ( path finding)**

Create a graph with the cities as nodes and the roads as arcs. To find a path of length \( nr \) from node A to B, look for a node C such that an arc exists from A to C and a path of length \( nr-1 \) exists from C to B. If these conditions are satisfied for some node C, the desired path exists, else path doesn’t exist. The algorithm uses an auxiliary recursive function `findpath(k,a,b)`. This function returns true if there is a path of length \( k \) from A to B and false otherwise. The algorithm for the program and function as follow:

```c
scanf("%d", &n); /* number of cities */
create n nodes and label them from 0 to n-1;
scanf("%d %d", &a,&b); /* seek path from a to b */
scanf("%d", &nr); /* desired number of roads to take */

while( scanf("%d%d", &city1, &city2) != EOF)
    join(city1,city2);

if(findpath(nr,a,b))
    printf("a path exists from %d to %d in %d steps", a, b, nr);
else
    printf("no path exists from %d to %d in %d steps", a, b, nr);

The algorithm for the function `findpath(k,a,b)` follows:

```c
if( k =1 )
    /* search for a path of length 1 */
    return ( adjacent (a,b) );
/* determine if there is a path through C */

for( c = 0 ; c < n ; ++c)
    if( adjacent ( a, c ) && findpath ( k-1, c, b )
        return ( TRUE );
    return ( FALSE ); /* assume no path exists */
```

**C Representation of Graphs :**

Suppose that the number of nodes in the graph is constant: that is, arcs may be added or deleted but nodes may not. A graph with 50 nodes could then be declared as follows:

```c
#define MAXNODES 50
struct node
{
    /* information associated with each node */
}

struct arc
{
    int adj; /* information associated with each arc */
};
```
struct graph
{
    struct node nodes[MAXNODES];
    struct arc arcs[MAXNODES][MAXNODES];
};
struct graph g;

The value of g.arcs[i][j].adj is either TRUE or FALSE depending on whether or not node j is adjacent to node i. The two-dimensional array g.arcs[][] .adj is called an **adjacency matrix**.

**Primitive Operations**:

For existence of arcs is declared simply by,

    int adj[MAXNODES][MAXNODES];

In effect, the graph is totally described by its adjacency matrix.

```c
join(adj, node1, node2)
int adj [ ] [MAXNODES];
int node1, node2;
{
    adj[node1][node2]=TRUE; /* add an arc from node1 to node2 */
}
remv(adj, node1, node2)
int adj [ ] [MAXNODES];
int node1, node2;
{
    adj[node1][node2]=FALSE; /* delete an arc from node1 to node2 */
}
adjacent(adj, node1, node2)
int adj [ ] [MAXNODES];
int node1, node2;
{
    return((adj[node1][node2]= = TRUE)? TRUE : FALSE);
}
```

A weighted graph with a fixed number of nodes may be declared by

```c
Struct arc
{
    int adj;
    int weight;
};
struct arc g[MAXNODES][MAXNODES];
```

The routine joinwt, which adds an arc from node1 to node2 with a given weight wt, may be coded as follows:
```c
jointwt(g, node1, node2, wt)
struct arc g[ ] [MAXNODES];
int node1, node2,w;  
{
  g[node1][node2].adj = TRUE;
  g[node1][node2].weight = wt;
}

|| the routine remvwt is coded.

Transitive Closure:

   Let us assume that a graph is completely described by its adjacency matrix, adj.  consider the logical expression adj[i][k] && adj[k][j].  Its value is TRUE if and only if the values of both adj[i][k] and adj[k][j] are TRUE, which implies that there is an arc from node i to node k and an arc from node k to node j.  ie, there is a path of length 2 from i to j passing through k.

   So, that the expression is,

   ( adj[i][0] && adj[0][j] ) || ( adj[i][1] && adj[1][j] ) || ........

   .... || ( adj[i][MAXNODES-1] && adj[MAXNODES][j] )

   Consider an array adj2 such that adj2[i][j] is the value of the foregoing expression.  adj2 is called the path matrix of length 2.

   adj2[i][j] indicates whether or not there is a path of length 2 between i to j.  adj2 is said to be the Boolean product of adj with itself.

||by

   Assume that path of length 3 or less exists between two nodes of a graph between i to j.  It must be of length 1,2 or 3 is written as follows:

   adj[i][j] || adj2[i][j] || adj3[i][j]

   Then, we wish to construct a matrix path of graph has node n such that path[i][j] is TRUE if and only if there is some path from node i to node j (of any length).  Clearly,

   Path[i][j] = = adj[i][j] || adj2[i][j] || .... || adjn[i][j].

   There must be another path from i to j of length less than or equal to n.  Note that there are only n nodes in the graph, at least one node k must appear in the path twice.  The path from i to j can be shortened by removing the cycle from k to k.  This process is repeated until no two nodes in the path (except i and j) are equal and therefore the path is of length n or less.  Such a path often called the transitive closure of the matrix adj.
```
We may write a C routine that accepts an adjacency matrix adj and computes its transitive closure path. This routine uses an auxiliary routine prod(a,b,c);

**transclose( adj, path)**

```c
int adj [ ] [MAXNODES], path [ ] [ MAXNODES];
{
    int i, j , k ;
    int newprod[MAXNODES], path [ ] [MAXNODES],
        adjprod[MAXNODES][MAXNODES];
    for(i=0 ; i < MAXNODES; ++i)
        for(j=0 ; j < MAXNODES; ++j)
            adjprod[i][j] = path[i][j] = ad[i][j];
    for(i=0 ; i < MAXNODES; ++i)
    {
        /* i represents the number of times adj has been multiplied by itself to obtain adjprod. At
         this point path represents all paths of length i or less */
        prod(adjprod, adj, newprod);
        for(j=0 ; j < MAXNODES; ++j)
            for(k=0 ; k < MAXNODES; ++k)
                path[j][k] = path[j][k] || newprod[j][k];
        for(j=0 ; j < MAXNODES; ++j)
            for(k=0 ; k < MAXNODES; ++k)
                adjprod[i][j] = newprod[j][k];
    }
}
```

The routine prod may be written as follows:

```c
prod( a, b , c)
int a[ ] [MAXNODES], b[ ] [MAXNODES], c[ ] [MAXNODES];
{
    int i, j, k, val;
    for(i=0 ; i < MAXNODES; ++i)  /* pass through rows */
        for(j=0 ; j < MAXNODES; ++j)  /* pass through columns */
        {
            val = FALSE;
            for(k=0 ; k < MAXNODES; ++k)
                val = val || ( a[i][k] && b[k][j]);
            c[i][j] = val;
        }
}
```