MOHANDAS COLLEGE OF ENGINEERING AND TECHNOLOGY

ANAD, NEDUMANGAD THIRUVANANTHAPURAM-695544

……………………………………………………….

LAB RECORD

Name: ………………………………………...

Reg No: ……………………………………....

Class: ………………**Branch:** ….…………

From: ………………**To:** ………………….

MOHANDAS COLLEGE
OF ENGINEERING AND TECHNOLOGY

ANAD, NEDUMANGAD THIRUVANANTHAPURAM-695544

………………………………………………….

LAB RECORD

Name: ……………………………………...

Reg No: ……………………………….....

Class: ………………**Branch:** ….…………

From: ………………**To:** ………………….

Certified Bonafide Record of Work Done By

……………………………………………………… **Place:**

…………………

Date: ………………….   **(Staff in Charge)**

Examiners

External   **Internal**

INDEX

SL.NO	NAME OF THE EXPERIMENT	DATE	PAGE NO	SIGNATURE

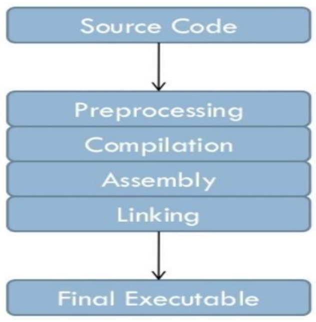
Experiment No :1

Date: 19-09-2023

Advanced GCC Commands In LINUX

GCC stands for GNU Compiler Collections. GCC Compiler is a very powerful and popular C compiler for various Linux distributions.

Compiling C files with gcc



GCC Compiler Options

Specify the Output Executable Name

Source File Name is main.c

In its most basic form, gcc compiler can be used as : gcc
main.c

The above command executes the complete compilation process and outputs an executable with name a.out.

Writes the build output to an output file using –o option.

Use option -o, as shown below, to specify the output file name for the executable.

Syntax:

```
gcc main.c -o main
```

The command above would produce an output file with name ‘main’.

Produce only the compiled code using the -c option

To produce only the compiled code (without any linking), use the -c option.

Syntax:

```
gcc -c main.c
```

The command above would produce a file main.o that would contain machine level code or the compiled code.

Use compile time macros using -D option

The compiler option D can be used to define compile time macros in code.

Syntax:

```
$ gcc -Dname [options] [source files] [-o output file]

$ gcc -Dname=definition [options] [source files] [-o output file]
```

Example

```
// main.c

#include <stdio.h> void main()
{
        #ifdef DEBUG printf("Debug
        run\n");
        #else

        printf("Release run\n"); #endif
}
```

Build *main.c* and run it with DEBUG defined:

```
$ gcc -D DEBUG main.c -o main

$ ./main
```

Or build *main.c* and run it without DEBUG defined:

```
$ gcc main.c -o main
$ ./main Release run
$
```

Link with shared libraries using -l option

The option -l can be used to link with shared libraries.

Syntax:

```
$ gcc [options] [source files] [object files] [-Ldir] -llibname [-o outfile]
```

Example:

```
$gcc -Wall main.c -o main –lCPPfile
```

The gcc command mentioned above links the code main.c with the shared library libCPPfile. So to produce the final executable ‘main’.

Include directory of header files using –I option

The option –I can be used to include directory of header files

Syntax:

```
$ gcc -Idir [options] [source files] [object files] [-o output file]
```

Example:

proj/src/myheader.h:

```
// myheader.h #define
NUM1 5
//main.c

#include <stdio.h> #include
"myheader.h" void main()
{
        int num = NUM1; printf("num=%d\n",
        num);
}
```

Debug information to be used by -g option

gcc -g generates debug information to be used by GDB debugger with 4 options –g0,-g1,-g,-g3

Syntax:

```
$ gcc -glevel [options] [source files] [object files] [-o output file]
```

```
$ gcc -g main.c -o main
$ gdb main
```

Setting the compiler optimization level using-O option

Set the compiler's optimization level several options such as -O0, -O1 ,-O2 ,-O3, - Os, -Ofast

Syntax:

```
$ gcc -Olevel [options] [source files] [object files] [-o output file]
```

```
$ gcc -O main.c -o main
```

Produce all the intermediate files using -save-temps function

Through this option, output at all the stages of compilation is stored in the current directory. This option produces the executable also.

Syntax:

```
$ gcc -save-temps source file
```

Example:

```
$ gcc -save-temps main.c

$ ls
```

```
a.out main.c main.i main.o main.s
```

main.i(preprocessed output) main.o(complied code) main.s(Assembly output)

GDB Commands in LINUX

The GNU Debugger, commonly abbreviated as GDB, is a command line tool that can be used to debug programs written in various programming languages such as C,C++,Ada, Fortran etc. The console can be using gdb command on terminal. gdb help us to find out watch or modify the variables in runtime, current state of the program, change the execute flow dynamically etc.

Compiling Programs with Debugging Information

To compile a C program with debugging information that can be read by the **GNU Debugger**, make sure the gcc compiler is run with the -g option.

Syntax:

$ gcc -g -o *output_file input_file*

Example:

```
$gcc -g -o main main.c
```

Common GDB Commands

run

"run" will start the program running under gdb. The program that starts will be the one that you have previously selected with the file command, or on the unix command line when you started gdb. You can give command line arguments to your program on the gdb command line without saying the program name. i.e., it runs the loaded executable program with program arguments arg1 ... argn.

Syntax:

[r]un (arg1 arg2 ... argn):

Example:

```
(gdb) run 2048 24 4
```

break

A ``breakpoint" is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the break command. It set a

breakpoint on either a function, a line given by a line number, or the instruction located at a particular address.
Syntax:

```
[b]reak <function name or file name: line# or *memory address>
```

Example:
```
(gdb) break main
```

Breakpoint 1 at 0x80488f6: file main.c, line 67.

next

"next" will continue until the next source line in the current function

Syntax:
```
[n]ext
```

Example:
```
(gdb) next
```

print

"print" *expression* will print out the value of the expression, which could be just a variable name.

Syntax:
```
[p]rint <expression>
```

Example:
```
(gdb) print list[0]@25
```

It prints out the first 25 values in an array called list

Display

Set an Automatic Display. It displays an expression every time the program stops.

Syntax:

```
display expression
```

Example:
```
(gdb) display/I $pc
```

It would display the next instruction after each step.

6. help

Gdb provides online documentation. Just typing help will give you a list of topics. Then you can type help *topic* /command to get information about that topic/command.

Syntax:

[h]elp [topic/command]

Example:

(gdb) help

gprof: LINUX GNU GCC Profiling Tool

Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. Through profiling one can determine the parts in program code that are time consuming and need to be re-written. This helps make your program execution faster which is always desired.

GNU profiling tool 'gprof' produces an execution profile of C, Pascal, or Fortran77 programs. `Gprof` calculates the amount of time spent in each routine. Next, these times are propagated along the edges of the call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle.

gprof is executing through the following steps:

Have profiling enabled while compiling the code

Execute the program code to produce the profiling data

Run the gprof tool on the profiling data file (generated in the step above).

Step-1 : Profiling enabled while compilation

In this first step, we need to make sure that the profiling is enabled when the compilation of the code is done. This is made possible by adding the '-pg' option in the compilation step. (-pg : Generate extra code to write profile information suitable for the analysis program gprof.)

```
$ gcc -pg testgprof.c new_testgprof.c -o testgprof
```

The option '-pg' can be used with the gcc command that compiles (-c option), gcc command that links(-o option on object files).

Step-2 : Execute the code

In the second step, the binary file produced as a result of step-1 is executed so that profiling information can be generated.

```
$ ./testgprof
```

If the profiling is enabled then on executing the program, file gmon.out (in built profile file for dynamic call graph and profile) will be generated.

```
$ ls
gmon.out testgprof testgprof.c
```

Step-3 : Run the gprof tool

In this step, the gprof tool is run with the executable name and the above generated 'gmon.out' as argument. This produces an analysis file which contains all the desired profiling information.

```
$ gprof testgprof gmon.out > analysis.txt
```

Experiment No: 9

Date: 25-11-2024

GRAPH TRAVERSAL TECHNIQUES :BFS, DFS, TOPOLOGICAL SORTING

AIM: To perform BFS , DFS , Topological sorting.

ALGORITHM:

BFS (Breadth-First Search)

Step 1.1: Initialize all vertices as unvisited (visited[i] = 0).

Step 1.2: Input the starting vertex v.

Step 1.3: Mark v as visited and enqueue it.

Step 1.4: While the queue is not empty:

Step 1.5: Dequeue a vertex, print it, and enqueue all its unvisited adjacent vertices.

Step 1.6: Repeat until all reachable vertices are visited.

DFS (Depth-First Search)

Step 2.1: Initialize all vertices as unvisited (reach[i] = 0).

Step 2.2: Input the starting vertex v.

Step 2.3: For each adjacent vertex of v:

Step 2.4: If unvisited, recursively visit the vertex and mark it as reached.

Step 2.5: Print all visited vertices.

Step 2.6: After traversal:

Step 2.7: Check if all vertices were reached.

Step 2.8: If all vertices are visited, the graph is connected. Otherwise, it is disconnected.

Topological Sort

Step 3.1: Calculate the in-degree for all vertices:

Step 3.2: For each vertex j, sum the values in column j of the adjacency matrix to

compute indegree[j].

Step 3.3: Push all vertices with in-degree 0 into a stack.

Step 3.4: While the stack is not empty:

Step 3.5: Pop a vertex u from the stack and add it to the topological order.

Step 3.6: For each adjacent vertex v of u:

Step 3.7: Decrement its in-degree.

Step 3.8: If indegree[v] == 0, push it onto the stack.

Step 3.9: Print the topological order.

Exit

Step 4.1: Exit the program.

<u>SOURCE CODE:</u>

```
#include<stdio.h>

#include<conio.h>

#define MAX 100

int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1,indegree[10];

int adj[MAX][MAX],reach[20];
int queue[MAX], front = -1,rear = -1;

void main()

{

  int ch,v,j,G[MAX][MAX],count=0;

  clrscr();

  do

  {

            Printf("\nGraph Traversal");

            printf("\n1 BFS\n2 DFS\n3 Topological Sort\n4 Exit \nEnter Choice:");

    scanf("%d",&ch);

    switch(ch)

    {

       case 1:

                printf("Enter the number of vertices:");

                scanf("%d",&n);

                for(i=1;i<=n;i++)

                {

                  q[i]=0;

                  visited[i]=0;

                }

                printf("\nEnter graph data in matrix form:\n");

                for(i=1;i<=n;i++)
```

```
        for(j=1;j<=n;j++)

        scanf("%d",&a[i][j]);

        printf("\nEnter the starting vertex:");

        scanf("%d",&v);

        visited[v]=1;

        printf("%d\t",v);

        bfs(v);

        break;

    case 2:

        printf("\n Enter the number of vertices:");

        scanf("%d",&n);

        for (i=1;i<=n;i++)

        {

        reach[i]=0;

        for (j=1;j<=n;j++)

        a[i][j]=0;

            }

            printf("\n Enter the adjacency matrix:\n");

            for (i=1;i<=n;i++)

        for (j=1;j<=n;j++)

        scanf("%d",&a[i][j]);

            dfs(1);
```

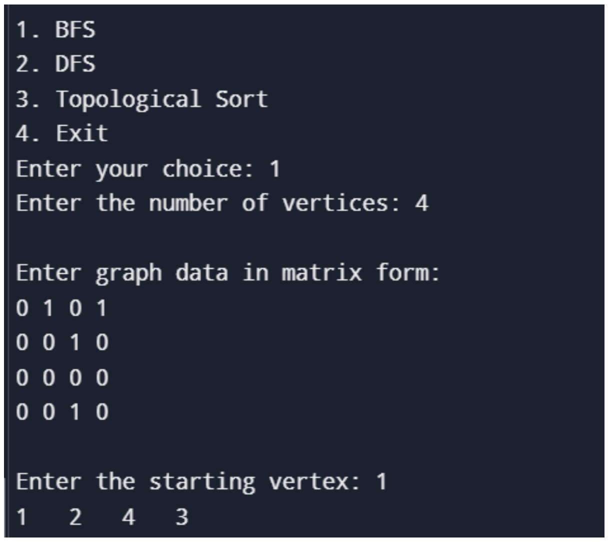
```
          printf("\n");
          for (i=1;i<=n;i++)
          {
      if(reach[i])
      count++;
          }
          if(count==n)
      printf("\n Graph is connected");
          else
      printf("\n Graph is not connected");
          break;
case 3:
      printf("\n Enter the number of vertices:");
      scanf("%d",&n);
      printf("Enter the adjacency matrix:\n ");
      for(i=0;i<n;i++)
      {
      for(j=0;j<n;j++)
      scanf("%d",&a[i][j]);
      }
      topology();
      getch();
```

```
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid input");
            }
    }while(ch!=5);
    getch();
}
int bfs(int v)
{
 for(i=1;i<=n;i++)
   if(a[v][i] && !visited[i])
     q[++r]=i;
   if(f<=r)
   {
   visited[q[f]]=1;
   printf("%d\t",q[f]);
   bfs(q[f++]);
   }
 return 0;
}
```

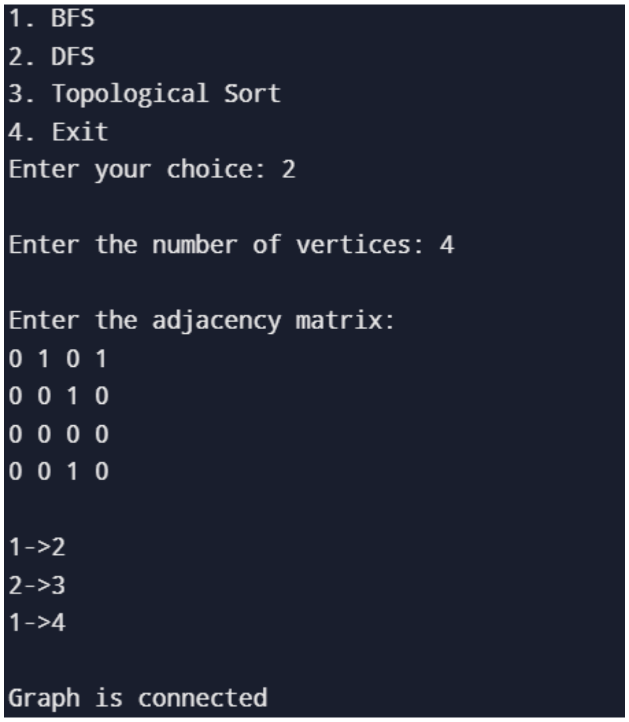
```
int dfs(int v)
{
 int i;
 reach[v]=1;
 for (i=1;i<=n;i++)
  if(a[v][i] && !reach[i])
  {
     printf("\n %d->%d",v,i);
     dfs(i);
}
     printf("\n%d",v);
return 0;
}
void find_indegree()
{
 int i,j,sum;
 for(j=0;j<n;j++)
 {
  sum=0;
  for(i=0;i<n;i++)
   sum+=a[i][j];
 indegree[j]=sum;
 }
}
int topology()
{
 int i,u,v,t[10],s[10],top=-1,k=0;
 delay(1000);
 find_indegree();
 for(i=0;i<n;i++)
 {
  if(indegree[i]==0)s[++top]=i;
 }
 while(top!=-1)
 {
  u=s[top--];
  t[k++]=u;
  for(v=0;v<n;v++)
  {
   if(a[u][v]==1)
   {
    indegree[v]--;
    if(indegree[v]==0)s[++top]=v;
   }
  }

 }

 printf("The Topological Sequence is:");

 for(i=0;i<n;i++)

 printf("\t%d",t[i]+1);

 return 0;

}
```

RESULT: Output is obtained and the result is verified.

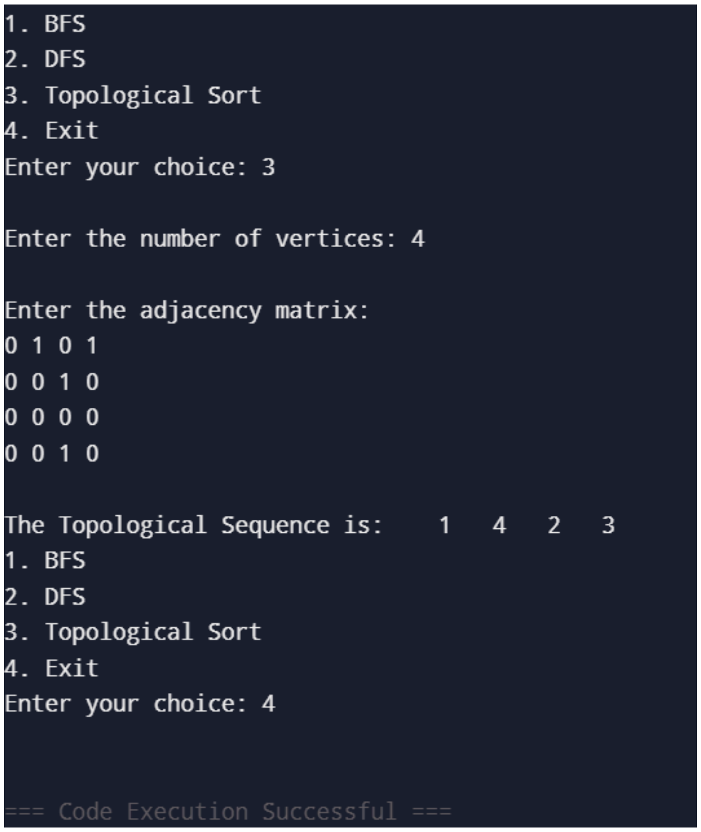
OUTPUT:



```
1. BFS
2. DFS
3. Topological Sort
4. Exit
Enter your choice: 1
Enter the number of vertices: 4

Enter graph data in matrix form:
0 1 0 1
0 0 1 0
0 0 0 0
0 0 1 0

Enter the starting vertex: 1
1   2   4   3
```



```
1. BFS
2. DFS
3. Topological Sort
4. Exit
Enter your choice: 2

Enter the number of vertices: 4

Enter the adjacency matrix:
0 1 0 1
0 0 1 0
0 0 0 0
0 0 1 0

1->2
2->3
1->4

Graph is connected
```



```
1. BFS
2. DFS
3. Topological Sort
4. Exit
Enter your choice: 3

Enter the number of vertices: 4

Enter the adjacency matrix:
0 1 0 1
0 0 1 0
0 0 0 0
0 0 1 0

The Topological Sequence is:    1   4   2   3
1. BFS
2. DFS
3. Topological Sort
4. Exit
Enter your choice: 4



=== Code Execution Successful ===
```

Experiment No: 10

Date: 02-12-2024

PRIM'S ALGORITHM

AIM: To perform minimum cost spanning tree using prim's algorithm

ALGORITHM:

1. Input:

Step 1.1: The number of nodes n in the graph.

Step 1.2: The adjacency matrix cost[][] where each element represents the cost of the edge between two nodes. If no edge exists, the value is set to 0 (which is replaced by a large number, typically 999 in your code).

2. Initialization:

Step 2.1: Mark the first node (node 1) as visited.

Step 2.2: Set the mincost variable to 0 to store the total weight of the MST.

Step 2.3: Set the variable ne (number of edges) to 1 (since the first edge will be added).

Step 2.4: Initialize the visited[] array to track the visited nodes, with visited[1] = 1 (first node is visited).

3. Process:

Step 3.1: Repeat the process until ne < n (i.e., until the number of edges in the MST is less than the number of nodes):

Step 3.2: For each node, if it is visited, check its adjacent nodes.

Step 3.3: Find the edge with the minimum cost from the already visited node to any unvisited node. This is the edge that should be added to the MST.

Step 3.4: Once the minimum cost edge is found, mark the node at the other end of the edge as visited and include it in the MST.

Step 3.5: Add the cost of this edge to the mincost and print the edge details (from node a to node b).

Step 3.6: Remove the edge from the cost matrix by setting its value to a very large number (like 999).

4. Termination:

Step 4.1: The algorithm stops when ne == n (the MST includes n-1 edges).

Step 4.2: Finally, print the total minimum cost (mincost) of the MST.

SOURCE CODE:

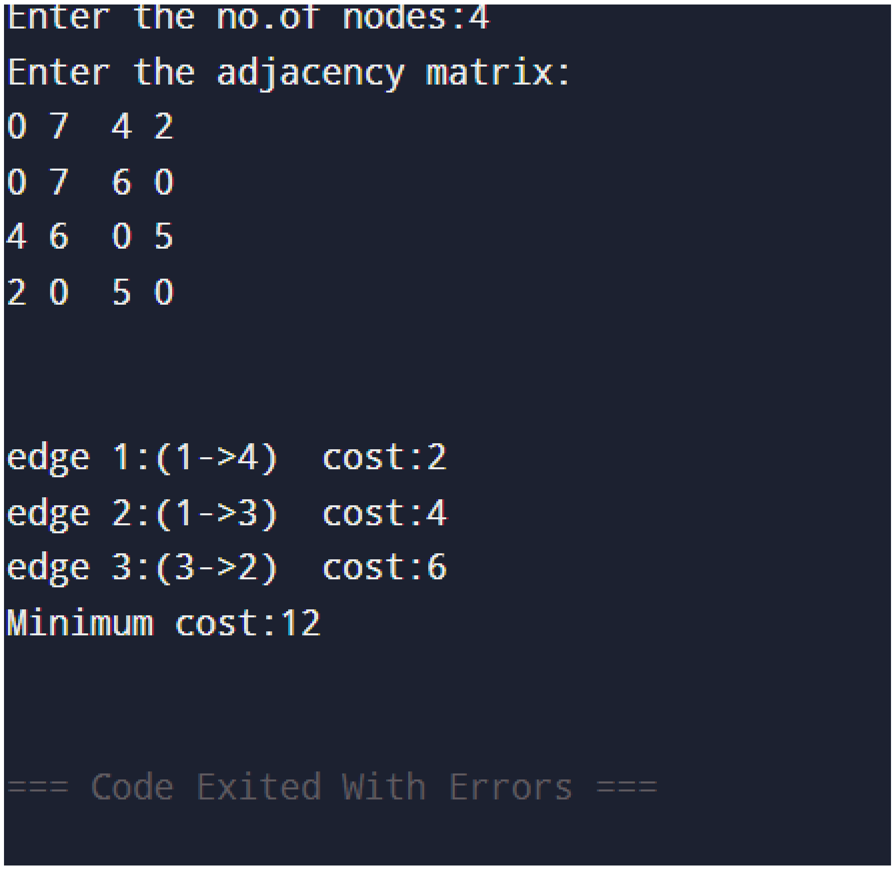
```
#include<stdio.h>

#include<conio.h>

int a,b,u,v,n,i,j,ne=1;

int visited[10]={0},min,mincost=0,cost[10][10];

void main()

 {

   clrscr();
```

```
printf("Enter the no.of nodes:");

scanf("%d",&n);

printf("Enter the adjacency matrix:\n");

for(i=1;i<=n;i++)

  for(j=1;j<=n;j++)

    {

           scanf("%d",&cost[i][j]);

           if(cost[i][j]==0)

               cost[i][j]=999;

    }

    visited[1]=1;

    printf("\n");

    while(ne<n)

         {

           for(i=1,min=999;i<=n;i++)

             for(j=1;j<=n;j++)

                if(cost[i][j]<min)

                       if(visited[i]!=0)

                         {

                            min=cost[i][j];

                            a=u=i;

                            b=v=j;
```

```
                    }
                    if(visited[u]==0 || visited[v]==0)
                     {
                       printf("\nedge %d:(%d->%d)  cost:%d",ne++,a,b,min);
                       mincost+=min;
                       visited[b]=1;
                     }
                     cost[a][b]=cost[b][a]=999;
        }
        printf("\nMinimum cost:%d\n",mincost);
        getch();
      }
```

RESULT: Output is obtained and the result is verified.

OUTPUT:



```
Enter the no.of nodes:4
Enter the adjacency matrix:
0 7  4 2
0 7  6 0
4 6  0 5
2 0  5 0


edge 1:(1->4)  cost:2
edge 2:(1->3)  cost:4
edge 3:(3->2)  cost:6
Minimum cost:12


=== Code Exited With Errors ===
```

Experiment No: 11

Date: 09-12-2024

KRUSKAL'S ALGORITHM

AIM: To perform minimum cost spanning tree using Kruskal's algorithm

ALGORITHM:

Step 1: **Initialization**:All necessary variables (i, j, k, a, b, u, v, n, nedge, min, mincost, cost[9][9], parent[9]) are initialized.

Step 2: **Input the number of vertices (n)**: The user is prompted to enter the number of vertices in the graph.

Step 3: **Input the adjacency matrix**: The user is asked to input the adjacency matrix cost[i][j], and any 0 values (representing no edge between nodes) are replaced with 999 (infinity).

Step 4: **Initialize the parent array**: The parent[] array is initialized to 0, meaning that initially, each vertex is its own parent.

Step 5: **Start the MST**: The process of finding the minimum spanning tree begins. The algorithm will loop until n-1 edges are added to the MST.

Step 6: **Loop through edges**: While the number of edges in the MST (nedge) is less than n-1, the algorithm continues.

Step 7: **Find the minimum weight edge**: The algorithm scans the cost[][] matrix to find the edge with the smallest weight that has not yet been selected.

Step 8: **Cycle check**: The find() function is used to check if adding the selected edge would form a cycle. If u and v belong to different sets, the edge is added to the MST.

Step 9: **Union operation**: If the edge (a, b) does not form a cycle, the uni() function joins the sets of u and v.

Step 10: **Remove the selected edge**: Once the edge is added to the MST, the corresponding entry in the adjacency matrix is set to 999 to avoid selecting it again.

Step 11: **Output the total minimum cost**: After the MST has been formed, the total minimum cost is displayed.

Step 11.1: **End program**: The program waits for user input before exiting.

Step 11.2: **Find function**: The find() function traces the parent array to find the root of a vertex. It returns the root of the set containing the vertex.

Step 11.3: **Union function**: The uni() function checks if two vertices belong to different sets. If they do, it unites the sets by making one vertex the parent of the other.

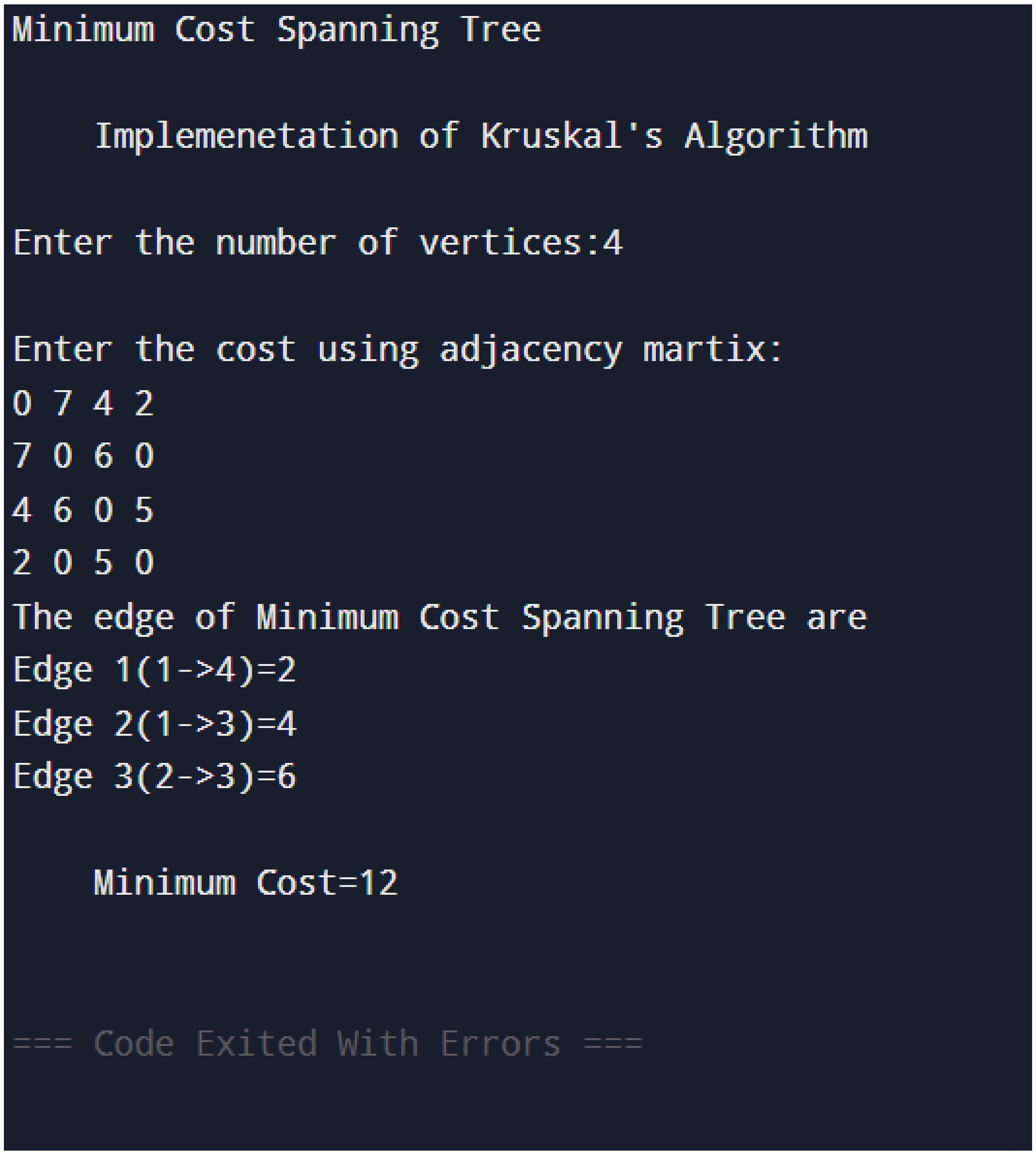
SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,nedge=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
	clrscr();
	printf("Minimum Cost Spanning Tree\n");
	printf("\n\tImplemenetation of Kruskal's Algorithm\n");
	printf("\nEnter the number of vertices:");
	scanf("%d",&n);
	printf("\nEnter the cost using adjacency martix:\n");
	for(i=1;i<=n;i++)
	{
		for(j=1;j<=n;j++)
		{
			scanf("%d",&cost[i][j]);
			if(cost[i][j]==0)
				cost[i][j]=999;

		}
	}
	printf("The edge of Minimum Cost Spanning Tree are\n");
	while(nedge<n)
	{
		for(i=1,min=999;i<=n;i++)
		{
			for(j=1;j<=n;j++)
			{
				if(cost[i][j]<min)
				{
					min=cost[i][j];
					a=u=i;
					b=v=j;
				}
			}
		}
		u=find(u);
		v=find(v);
		if(uni(u,v))
		{
			printf("Edge %d(%d->%d)=%d\n",nedge++,a,b,min);
			mincost+=min;
		}
		cost[a][b]=cost[b][a]=999;
	}
	printf("\n\tMinimum Cost=%d\n",mincost);
	getch();
}
int find(int i)
```

```
{
        while(parent[i])
                i=parent[i];
        return i;
}
int uni(int i,int j)
{
        if(i!=j)
        {
                parent[j]=i;
                return 1;
        }
        return 0;
}
```

RESULT: Output is obtained and the result is verified.

OUTPUT:



```
Minimum Cost Spanning Tree

    Implemenetation of Kruskal's Algorithm

Enter the number of vertices:4

Enter the cost using adjacency martix:
0 7 4 2
7 0 6 0
4 6 0 5
2 0 5 0
The edge of Minimum Cost Spanning Tree are
Edge 1(1->4)=2
Edge 2(1->3)=4
Edge 3(2->3)=6

    Minimum Cost=12



=== Code Exited With Errors ===
```

Experiment No: 12

Date: 17-12-2024

SINGLE SOURCE SHORTEST PATH : DIJKSTRA'S ALGORITHM

AIM: To find the single source shortest path algorithm.

ALGORITHM:

```
STEP 1 : Create a set shortPath to store vertices that come in the way of the shortest
         path tree.
STEP 2 : Initialize all distance values as INFINITE and assign distance values as 0 for
         source vertex so that it is picked first.
STEP 3 :  Loop until all vertices of the graph are in the shortPath.
STEP 3.1 : Take a new vertex that is not visited and is nearest.
STEP 3.2 : Add this vertex to shortPath.
STEP 3.3 : For all adjacent vertices of this vertex update distances. Now check
           every adjacent vertex of V, if sum of distance of u and weight of edge is
           else then update it.
```

SOURCE CODE:

```
#include<stdio.h>

#include<conio.h>

#define INFINITY 9999

#define MAX 10
```

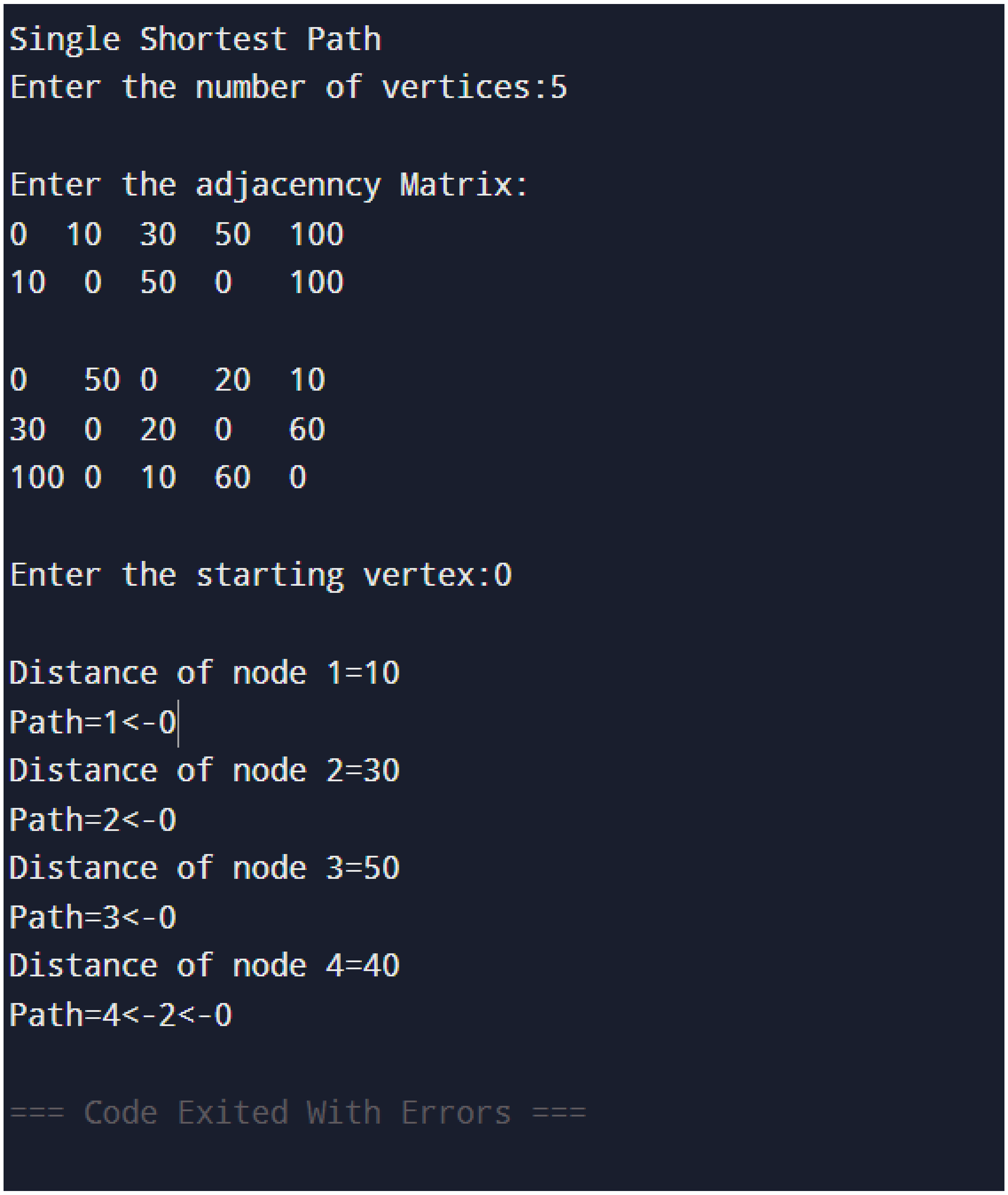
```
void dijkstra(int g[MAX][MAX],int n,int startnode);

void main()

{

        int g[MAX][MAX],i,j,n,u;

        clrscr();

        printf("\nSingle Shortest Path\n");

        printf("Enter the number of vertices:");

        scanf("%d",&n);

        printf("\nEnter the adjacenncy Matrix:\n");

        for(i=0;i<n;i++)

                for(j=0;j<n;j++)

                        scanf("%d",&g[i][j]);

        printf("\nEnter the starting vertex:");

        scanf("%d",&u);

        dijkstra(g,n,u);

        getch();

}

void dijkstra(int g[MAX][MAX],int n, int startnode)

{

        int cost[MAX][MAX],distance[MAX],pred[MAX];

        int visited[MAX],count,mindistance,nextnode,i,j;

        for(i=0;i<n;i++)
                for(j=0;j<n;j++)

                {

                        if(g[i][j]==0)

                                cost[i][j]=INFINITY;

                        else

                                cost[i][j]=g[i][j];

                }

        for(i=0;i<n;i++)

        {

                distance[i]=cost[startnode][i];

                pred[i]=startnode;

                visited[i]=0;

        }

        distance[startnode]=0;

        visited[startnode]=1;

        count=1;

        while(count<n-1)

        {

                mindistance=INFINITY;

                for(i=0;i<n;i++)

                        if(distance[i]<mindistance && !visited[i])

                        {
```

```
                mindistance=distance[i];
                nextnode=i;
            }
        visited[nextnode]=1;
        for(i=0;i<n;i++)
            if(!visited[i])
                if(mindistance+cost[nextnode][i]<distance[i])
                {
                    distance[i]=mindistance+cost[nextnode][i];
                    pred[i]=nextnode;
                }
        count++;
    }
    for(i=0;i<n;i++)
        if(i!=startnode)
        {
            printf("\nDistance of node %d=%d",i,distance[i]);
            printf("\nPath=%d",i);
            j=i;
            do
            {
                j=pred[j];
```

```
                        printf("<-%d",j);

                    }while(j!=startnode);

                }

        }
```

RESULT: Output is obtained and the result is verified.

OUTPUT:



```
Single Shortest Path
Enter the number of vertices:5

Enter the adjacenncy Matrix:
0   10  30  50  100
10  0   50  0   100

0   50 0    20  10
30  0   20  0   60
100 0   10  60  0

Enter the starting vertex:0

Distance of node 1=10
Path=1<-0
Distance of node 2=30
Path=2<-0
Distance of node 3=50
Path=3<-0
Distance of node 4=40
Path=4<-2<-0


=== Code Exited With Errors ===
```