

EXPERIMENT NO- 11

AIM: Program to compute the transitive closure of a given graph using Warshall's algorithm.

PROBLEM STATEMENT: Write a program to compute the transitive closure of a given graph using Warshall's algorithm.

RESOURCE REQUIRED: Pentium IV, Turbo C, Printer, Printout Stationary

THEORY:

Warshall's Algorithm to find Transitive Closure: -

Suppose we have a directed graph $G = (V, E)$. It's useful to know, given a pair of vertices u and w , whether there is a path from u to w in the graph. A nice way to store this information is to construct another graph, call it $G^* = (V, E^*)$, such that there is an edge (u, w) in G^* if and only if there is a path from u to w in G . This graph is called the transitive closure of G .

The name "transitive closure" means this:

- Having the transitive property means that if a is related to b in some special way, and b is related to c , then a is related to c . You are familiar with many forms of transitivity. For example, the "less than" operator is transitive for real numbers: if $a < b$ and $b < c$, then $a < c$. In the case of graphs, we say a graph is transitive if, for every triple of vertices a , b , and c , if (a, b) is an edge, and (b, c) is an edge, then (a, c) is also an edge. Some graphs are transitive, some graphs aren't.
- A set A^* is a closure of a set A with respect to some special property (like transitivity) is the result adding to A only the elements that cause A to satisfy that special property, and no other elements. (Nothing is taken away from A). For example, let's take a property like "additiveness". This means that if a and b are in a set, then $a+b$ should also be in the set. Then the "additive closure" of, for example, $\{2\}$, would be the set of even numbers, the additive closure of $\{1, -1\}$ would be the set of integers, and so forth. The transitive closure of a graph is the result of adding the fewest possible edges to the graph such that it is transitive. (We can easily add a bunch of edges to a graph to make it transitive, but the closure part means we want to preserve path relationships that existed before, i.e., don't add edges that don't represent paths in the graph.)

How can we compute the transitive closure of a graph? One way is to run Dijkstra's Algorithm on each vertex, placing an edge (u, w) in the transitive closure if there the shortest path from u to w isn't of infinite length (i.e., exists). If there are n vertices in the graph, Dijkstra's Algorithm takes $O(n^2 \log n)$ time using a heap-based priority queue. Running the algorithm n times would take $O(n^3 \log n)$ time. We'd like to do better than this.

We'll represent graphs using an adjacency matrix of Boolean values. We'll call the matrix for our graph G $t(0)$, so that $t(0)[i, j] = \text{True}$ if there is an edge from vertex i to vertex j OR if $i=j$, False otherwise. (This last bit is an important detail; even though, with standard definitions of graphs, there is never an edge from a vertex to itself, there is a path, of length 0, from a vertex to itself.)

Let n be the size of V . For k in $0 \dots n$, let $t(k)$ be an adjacency matrix such that, if there is a path in G from any vertex i to any other vertex j going only through vertices in $\{1, 2, \dots, k\}$, then $t(k)[i, j] = \text{True}$, False otherwise.

This set $\{1, 2, \dots, k\}$ contains the intermediate vertices along the path from one vertex to another. This set is empty when $k=0$, so our previous definition of $t(0)$ is still valid.

When $k=n$, this is the set of all vertices, so $t(n)[i,j]$ is True if and only if there is a path from i to j through any vertex. Thus $t(n)$ is the adjacency matrix for the transitive closure of G . Now all we need is a way to get from $t(0)$, the original graph, to $t(n)$, the transitive closure. Consider the following rule for doing so in steps, for $k \geq 1$:

$$t(k)[i,j] = t(k-1)[i,j] \text{ OR } (t(k-1)[i,k] \text{ AND } t(k-1)[k,j])$$

In English, this says $t(k)$ should show a path from i to j if

1. $t(k-1)$ already shows a path from i to j , going through one of the vertices in $1..k-1$, OR
2. $t(k-1)$ shows a path from i to k AND a path from k to j ; this way, we can go from i to j through k .

So to find $t(n)$, the transitive closure, we just let k start at 0 and then apply this rule iteratively to get $t(1)$, $t(2)$, $t(3)$, ... until $k=n$ and we get $t(k)$. Here's the algorithm. Your book calls it Transitive-Closure, but it is commonly known in the literature as "Warshall's Algorithm."

Transitive-Closure (G) $n = |V|$

$t(0)$ = the adjacency matrix for G

// there is always an empty path from a vertex to itself,
// make sure the adjacency matrix reflects this

```
for i in 1..n do
  t(0)[i,i] = True
end for
// step through the t(k)'s for k in 1..n do
  for i in 1..n do
    for j in 1..n do
      t(k)[i,j] = t(k-1)[i,j] OR
      (t(k-1)[i,k] AND t(k-1)[k,j])
```

```
end for
```

```
end for
```

```
end for
```

```
return t(n)
```

This algorithm simply applies the rule n times, each time considering a new vertex through which possible paths may go. At the end, all paths have been discovered.

Let's look at an example of this algorithm. Consider the following graph:

So we have $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 1), (3, 6), (4, 6), (4, 3), (6, 5)\}$. Here is the adjacency matrix and corresponding $t(0)$:

down = "from" across = "to"

adjacency matrix for G: $t(0)$:

1 2 3 4 5 6	1 2 3 4 5 6
1 0 1 1 0 0 0	1 1 1 1 0 0 0
2 0 0 0 1 1 0	2 0 1 0 1 1 0
3 1 0 0 0 0 1	3 1 0 1 0 0 1
4 0 0 1 0 0 1	4 0 0 1 1 0 1
5 0 0 0 0 0 0	5 0 0 0 0 1 0
6 0 0 0 0 1 0	6 0 0 0 0 1 1

Now let's look at what happens as we let k go from 1 to 6: $k = 1$

add (3,2); go from 3 through 1 to 2

$t(1) =$

1 2 3 4 5 6
1 1 1 1 0 0 0
2 0 1 0 1 1 0
3 1 1 1 0 0 1
4 0 0 1 1 0 1
5 0 0 0 0 1 0
6 0 0 0 0 1 1

$k = 2$

add (1,4); go from 1 through 2 to 4

add (1,5); go from 1 through 2 to 5

add (3,4); go from 3 through 2 to 4

add (3,5); go from 3 through 2 to 5 $t(2) =$

1 2 3 4 5 6
1 1 1 1 1 1 0
2 0 1 0 1 1 0
3 1 1 1 1 1 1
4 0 0 1 1 0 1
5 0 0 0 0 1 0
6 0 0 0 0 1 1

$k = 3$

add (1,6); go from 1 through 3 to 6

add (4,1); go from 4 through 3 to 1

add (4,2); go from 4 through 3 to 2

add (4,5); go from 4 through 3 to 5 $t(3) =$

1 2 3 4 5 6
1 1 1 1 1 1 1
2 0 1 0 1 1 0
3 1 1 1 1 1 1
4 1 1 1 1 1 1
5 0 0 0 0 1 0

6 0 0 0 0 1 1

k = 4

add (2,1); go from 2 through 4 to 1

add (2,3); go from 2 through 4 to 3

add (2,6); go from 2 through 4 to 6 t(4) =

1 2 3 4 5 6

1 1 1 1 1 1

2 1 1 1 1 1

3 1 1 1 1 1

4 1 1 1 1 1

5 0 0 0 0 1 0

6 0 0 0 0 1 1

k = 5 t(5) =

1 2 3 4 5 6

1 1 1 1 1 1

2 1 1 1 1 1

3 1 1 1 1 1

4 1 1 1 1 1

5 0 0 0 0 1 0

6 0 0 0 0 1 1

k = 6 t(6) =

1 2 3 4 5 6

1 1 1 1 1 1

2 1 1 1 1 1

3 1 1 1 1 1

4 1 1 1 1 1

5 0 0 0 0 1 0

6 0 0 0 0 1 1

At the end, the transitive closure is a graph with a complete subgraph (a clique) involving vertices 1, 2, 3, and 4. You can get to 5 from everywhere, but you can get nowhere from 5. You can get to 6 from everywhere except for 5, and from 6 only to 5.

CODE:

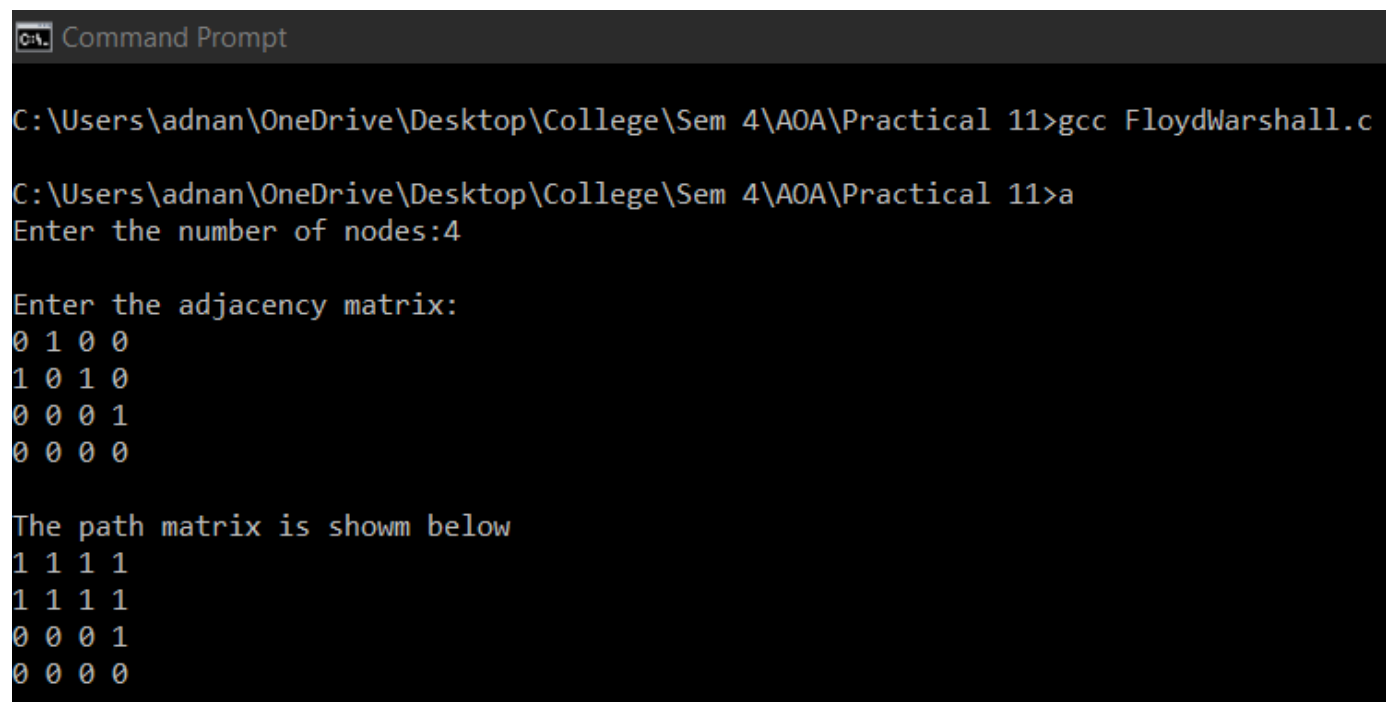
```
# include <stdio.h>
# include <conio.h>
int n,a[10][10],p[10][10];
void path()
{
    int i,j,k;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            p[i][j]=a[i][j];
    for(k=0;k<n;k++)
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                if(p[i][k]==1&& p[k][j]==1) p[i][j]=1;
}
void main()
{
```

```
int i,j;

printf("Enter the number of nodes:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&a[i][j]);
path();
printf("\nThe path matrix is showm below\n");
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
printf("%d ",p[i][j]);
printf("\n");
}

}
```

OUTPUT:



```
Command Prompt

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\AOA\Practical 11>gcc FloydWarshall.c

C:\Users\adnan\OneDrive\Desktop\College\Sem 4\AOA\Practical 11>a
Enter the number of nodes:4

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

The path matrix is showm below
1 1 1 1
1 1 1 1
0 0 0 1
0 0 0 0
```

CONCLUSION: This algorithm has three nested loops containing a (1) core, so it takes (n^3) time complexity. we would need (n^3) storage; however, note that at any point in the algorithm, we only need the last two matrices computed, so we can re-use the storage from the other matrices, bringing the storage complexity down to (n^2).