EXPERIMENT-7

OBJECTIVE

Write an Echo_server using TCP to estimate the round-trip time from client to the server. The server should be such that it can accept multiple connections at any given time, with multiplexed I/O operations.

ALGORITHM:

//Echo server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include<string.h>
int main()
{

charstr[100];
intlisten_fd, comm_fd;

structsockaddr_inservaddr;

listen_fd = socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htons(INADDR_ANY);
servaddr.sin_port = htons(22000);

bind(listen_fd, (structsockaddr *) &servaddr, sizeof(servaddr));

listen(listen_fd, 10);
comm_fd = accept(listen_fd, (structsockaddr*) NULL, NULL);

while(1)
   {

bzero(str, 100);

read(comm_fd,str,100);

printf("Echoing back - %s",str);
```

```
write(comm_fd, str, strlen(str)+1);

    }
}
```

To Run:
$ gcc –o server echoserver.c
$ ./server

Output:

Output:
Echoing back – Hello
Echoing back – How r u
Echoing back – I m fine
Echoing back – Bye

//**Client process**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include<string.h>
int main(intargc,char **argv)
{
intsockfd,n;
charsendline[100];
charrecvline[100];
structsockaddr_inservaddr;

sockfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeofservaddr);

servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(22000);

inet_pton(AF_INET,"127.0.0.1",&(servaddr.sin_addr));

connect(sockfd,(structsockaddr *)&servaddr,sizeof(servaddr));

while(1)
    {
bzero(sendline, 100);
bzero(recvline, 100);
fgets(sendline,100,stdin); /*stdin = 0 , for standard input */
```

```
write(sockfd,sendline,strlen(sendline)+1);
read(sockfd,recvline,100);
printf("%s",recvline);
    }

}
```

To Run:

```
$ gcc –o client client1.c
$ ./client 22000
```

OUTPUT

```
Hello
Hello
How r u
How r u
I m fine
I m fine
Bye
Bye
```

EXPERIMENT-8

OBJECTIVE

Write a program in C: hello_client (The server listens for, and accepts, a single UDP connection; it reads all the data itcan from that connection, and prints it to the screen; then it closes the connection)

PROGRAM

```
// UDP Client
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

#define MAX_BUF 100

intmain(intargc,char*argv[])
{
intsockd;
structsockaddr_inmy_addr,srv_addr;
charbuf[MAX_BUF];
int count;
intaddrlen;

if(argc<3)
{
fprintf(stderr,"Usage: %s ip_addressport_number\n",argv[0]);
exit(1);
}
/* create a socket */
sockd=socket(AF_INET, SOCK_DGRAM,0);
if(sockd==-1)
{
perror("Socket creation error");
exit(1);
```

```
}

/* client address */
my_addr.sin_family= AF_INET;
my_addr.sin_addr.s_addr= INADDR_ANY;
my_addr.sin_port=0;

bind(sockd,(structsockaddr*)&my_addr,sizeof(my_addr));

strcpy(buf,"Hello world\n");

/* server address */
srv_addr.sin_family= AF_INET;
inet_aton(argv[1],&srv_addr.sin_addr);
srv_addr.sin_port=htons(atoi(argv[2]));

sendto(sockd,buf,strlen(buf)+1,0,
(structsockaddr*)&srv_addr,sizeof(srv_addr));

addrlen=sizeof(srv_addr);
count=recvfrom(sockd,buf, MAX_BUF,0,
(structsockaddr*)&srv_addr,&addrlen);
write(1,buf, count);

close(sockd);
return0;
```

EXPERIMENT- 09

OBJECTIVE

Write a programs in C: hello_server
(The client connects to the server, sends the string “Hello, world!”, then closes the UDP connection)

PROGRAM

//UDP Server

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

#define MAX_BUF 1024

intmain(intargc,char*argv[])
{
intsockd;
structsockaddr_inmy_name,cli_name;
charbuf[MAX_BUF];
int status;
intaddrlen;

if(argc<2)
{
fprintf(stderr,"Usage: %s port_number\n",argv[0]);
exit(1);
}
/* create a socket */
sockd=socket(AF_INET, SOCK_DGRAM,0);
if(sockd==-1)
{
perror("Socket creation error");
```

```
exit(1);
}

/* server address */
my_name.sin_family= AF_INET;
my_name.sin_addr.s_addr= INADDR_ANY;
my_name.sin_port=htons(atoi(argv[1]));

status=bind(sockd,(structsockaddr*)&my_name,sizeof(my_name));

addrlen=sizeof(cli_name);
status=recvfrom(sockd,buf, MAX_BUF,0,
(structsockaddr*)&cli_name,&addrlen);

printf("%s",buf);
strcat(buf,"OK!\n");

status=sendto(sockd,buf,strlen(buf)+1,0,
(structsockaddr*)&cli_name,sizeof(cli_name));

close(sockd);
return0;
}
```

Echoing back – Hello
Echoing back – How r u
Echoing back – I m fine
Echoing back – Bye

//**Client process**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include<string.h>
int main(intargc,char **argv)
{
intsockfd,n;
charsendline[100];
charrecvline[100];
structsockaddr_inservaddr;
```

```
sockfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeofservaddr);

servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(22000);

inet_pton(AF_INET,"127.0.0.1",&(servaddr.sin_addr));

connect(sockfd,(structsockaddr *)&servaddr,sizeof(servaddr));

while(1)
    {
bzero(sendline, 100);
bzero(recvline, 100);
fgets(sendline,100,stdin); /*stdin = 0 , for standard input */

write(sockfd,sendline,strlen(sendline)+1);
read(sockfd,recvline,100);
printf("%s",recvline);
    }

}
```

To Run:

```
$ gcc –o client client1.c
$ ./client 22000
```

Output:

```
Hello
Hello
How r u
How r u
I m fine
I m fine
Bye
Bye
```

Lab outcome: By this experiment student will be able to create TCP echo client and server processes. This satisfies LO[4].

EXPERIMENT-10

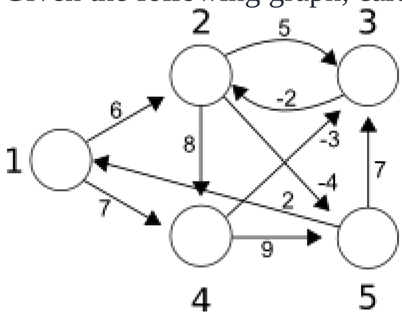
OBJECTIVE

Program to simulate Bellman Ford Routing Algorithm

PROGRAM

The Problem

Given the following graph, calculate the length of the shortest path from **node 1** to **node 2**.



It’s obvious that there’s a direct route of length 6, but take a look at path: 1 -> 4 -> 3 -> 2. The length of the path is 7 – 3 – 2 = 2, which is less than 6. BTW, you don’t need negative edge weights to get such a situation, but they do clarify the problem.

This also suggests a property of shortest path algorithms: to find the shortest path form xto y, you need to know, beforehand, the shortest paths to y‘s neighbours. For this, you need to know the paths to y‘s neighbours’neighbours… In the end, you must calculate the shortest path to the connected component of the graph in which x and y are found.

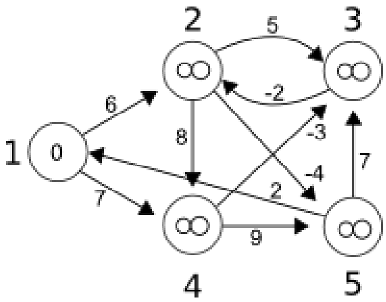
That said, you usually calculate **the shortest path to all nodes** and then pick the ones you’re interested in.

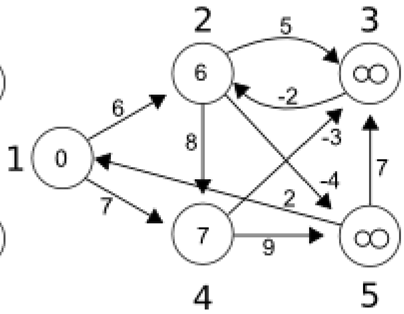
The Algorithm

The Bellman-Ford algorithm is one of the classic solutions to this problem. It calculates the shortest path to all nodes in the graph from a single source.

The basic idea is simple:

Start by considering that the shortest path to all nodes, less the source, is infinity. Mark the length of the path to the source as 0:

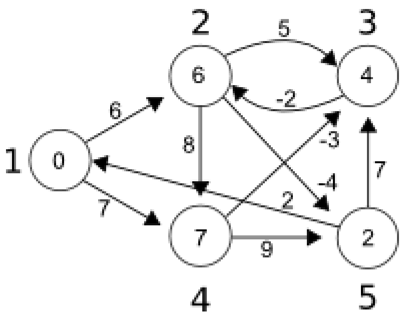


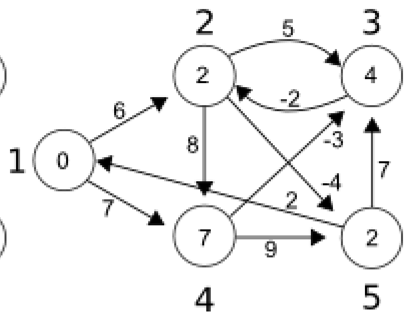


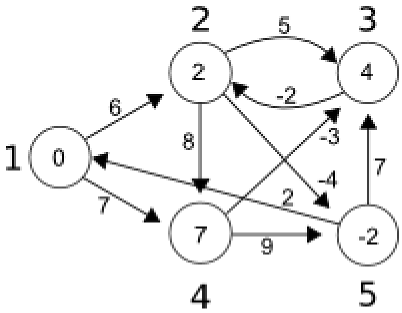
Take every edge and try to relax it:

Relaxing an edge means checking to see if the path to the node the edge is pointing to can't be shortened, and if so, doing it. In the above graph, by checking the **edge 1 -> 2** of length 6, you find that the length of the shortest path to **node 1** plus the length of the**edge 1 -> 2** is less then infinity. So, you replace infinity in **node 2** with 6. The same can be said for edge 1 -> 4 of length 7. It's also worth noting that, practically, you can't relax the edges whose start has the shortest path of length infinity to it.

Now, you apply the previous step n – 1 times, where n is the number of nodes in the graph. In this example, you have to apply it 4 times (that's 3 more times).







Here, **d[i]** is the shortest path to node **i**, **e** is the number of edges and **edges[i]** is the **i**-th edge. It may not be obvious why this works, but take a look at what is certain after each step. After the first step, any path made up of at most 2 nodes will be optimal. After the step 2, any path made up of at most 3 nodes will be optimal… After the (n – 1)-th step, any path made up of at most n nodes will be optimal.

The Programme

The following programme just puts the **bellman_ford** function into context. It runs in**O(VE)** time, so for the example graph it will do something on the lines of **5 * 9 = 45**relaxations. Keep in mind that this algorithm works quite well on graphs with few edges, but is very slow for dense graphs (graphs with almost n^2 edges)

```
#include <stdio.h>

typedefstruct {
	int u, v, w;
} Edge;

int n; /* the number of nodes */
int e; /* the number of edges */
Edge edges[1024]; /* large enough for n <= 2^5=32 */
int d[32]; /* d[i] is the minimum distance from node s to node i */

#define INFINITY 10000

voidprintDist() {
	int i;

	printf("Distances:\n");

	for (i = 0; i < n; ++i)
		printf("to %d\t", i + 1);
	printf("\n");

	for (i = 0; i < n; ++i)
		printf("%d\t", d[i]);

	printf("\n\n");
}

voidbellman_ford(int s) {
	int i, j;

	for (i = 0; i < n; ++i)
		d[i] = INFINITY;

	d[s] = 0;

	for (i = 0; i < n - 1; ++i)
		for (j = 0; j < e; ++j)
			if (d[edges[j].u] + edges[j].w < d[edges[j].v])
				d[edges[j].v] = d[edges[j].u] + edges[j].w;
}
```

```
int main(intargc, char *argv[]) {
        int i, j;
        int w;

        FILE *fin = fopen("dist.txt", "r");
        fscanf(fin, "%d", &n);
        e = 0;

        for (i = 0; i < n; ++i)
                for (j = 0; j < n; ++j) {
                        fscanf(fin, "%d", &w);
                        if (w != 0) {
                                edges[e].u = i;
                                edges[e].v = j;
                                edges[e].w = w;
                                ++e;
                        }
                }
        fclose(fin);

        /* printDist(); */

        bellman_ford(0);

        printDist();

        return 0;
}
```

OUTPUT

And here’s the input file used in the example (dist.txt):

```
5
0 6 0 7 0
0 0 5 8 -4
0 -2 0 0 0
0 0 -3 9 0
2 0 7 0 0
```

--this is a nonblocking I/O--