

ANALYSIS OF ALGORITHMS 2

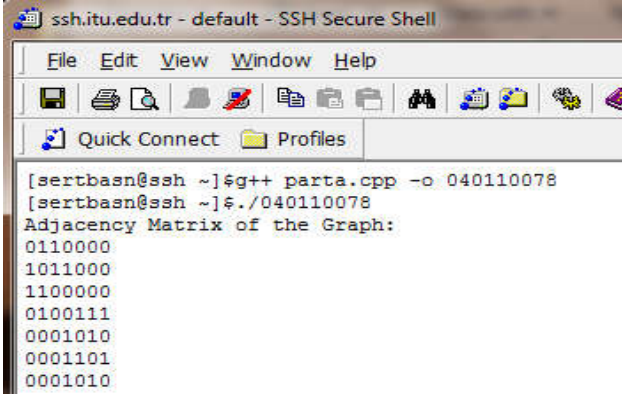
PROJECT 1

NurefşanSertbaş
040110078
sertbasn@itu.edu.tr

***Note:** The project contains 3 cpp file in order to implement part a, b and c seperately.

A. IMPLEMENTATION

First I compiled and executed my code in SSH in order to see if there exist any error. The sample execution output is shown in below figure.



```
ssh.itu.edu.tr - default - SSH Secure Shell
File Edit View Window Help
[sertbasn@ssh ~]$g++ parta.cpp -o 040110078
[sertbasn@ssh ~]$./040110078
Adjacency Matrix of the Graph:
0110000
1011000
1100000
0100111
0001010
0001101
0001010
```

It first reads input1.txt file in a suitable form and prints the adjacency matrix to the screen. Then, it creates graph by inserting edges and starts to do desired work over the graph.

Part a

In this part, we run our algorithm for 3 cases in order to list all possible paths from the source node to the termination node. Then, we select the shortest one(s) as a shortest path. It is implemented similar idea with the Dijkstra algorithm.

For node A to node F:

```
All possible paths from A to F
Path 1: A->B->D->E->F      Path Length:4
Path 2: A->B->D->F          Path Length:3
Path 3: A->B->D->G->F       Path Length:4
Path 4: A->C->B->D->E->F       Path Length:5
Path 5: A->C->B->D->F       Path Length:4
Path 6: A->C->B->D->G->F       Path Length:5
*** **
Shortest Path length:3
Shortest Path(s):
Path 2
*** **
[serbasn@ssh ~]$
```

For node E to node G:

```
All possible paths from E to G
Path 1: E->D->F->G          Path Length:3
Path 2: E->D->G              Path Length:2
Path 3: E->F->D->G          Path Length:3
Path 4: E->F->G              Path Length:2
*** **
Shortest Path length:2
Shortest Path(s):
Path 4
Path 2
*** **
[serbasn@ssh ~]$
```

For node B to node F:

```
All possible paths from B to F
Path 1: B->D->E->F          Path Length:3
Path 2: B->D->F              Path Length:2
Path 3: B->D->G->F          Path Length:3
*** **
Shortest Path length:2
Shortest Path(s):
Path 2
*** **
[serbasn@ssh ~]$
```

Part b

In this part, we have followed simple procedure in order to find betweenness values between given two nodes in the graph that read from input1.txt file. I compiled and executed my code as can be seen in below figure.



```
ssh.itu.edu.tr - default - SSH Secure Shell
File Edit View Window Help
[sertbasn@ssh ~]$ g++ partb.cpp -o 040110078
[sertbasn@ssh ~]$ clear all
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
AC
Betweenness of(C,A):1*1=1
[sertbasn@ssh ~]$ clear all
[sertbasn@ssh ~]$ g++ partb.cpp -o 040110078
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
AB
Betweenness of(B,A):5*1=5
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
AC
Betweenness of(C,A):1*1=1
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
BC
Betweenness of(C,B):1*5=5
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
BD
Betweenness of(D,B):4*3=12
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
DE
Betweenness of(E,D):1*5=5
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
DF
Betweenness of(F,D):1*4=4
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
DG
Betweenness of(G,D):1*5=5
```

```
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
EF
Betweenness of (F,E):2*1=2
[sertbasn@ssh ~]$ ./040110078
Enter two nodes to computes betweenness (ex: AB)
GF
Betweenness of (F,G):2*1=2
[sertbasn@ssh ~]$
```

Connected to ssh.itu.edu.tr

Outputs are explained with details in Report part b.

Part c

In this part, we have implement the breadth first search algorithm. Then, run it for the original graph defined in input2.txt file. After that, we have reversed all of the edges of the graph and print them in order to check its accuracy. We run BFS algorithm again for the reversed graph in this case.

```
[sertbasn@ssh ~]$ g++ partc.cpp -o 040110078
[sertbasn@ssh ~]$ ./040110078
Adjacency Matrix of the Graph:
01000
00001
01010
10100
00110
Graph is reversed new edges are:
1-0
4-1
1-2
3-2
0-3
2-3
2-4
3-4
BFS for graph:
A B E C D
BFS for reversed graph:
A D C E B
5nodes in g and 5 nodes in grev are visited
Graph is strongly connected
[sertbasn@ssh ~]$
```

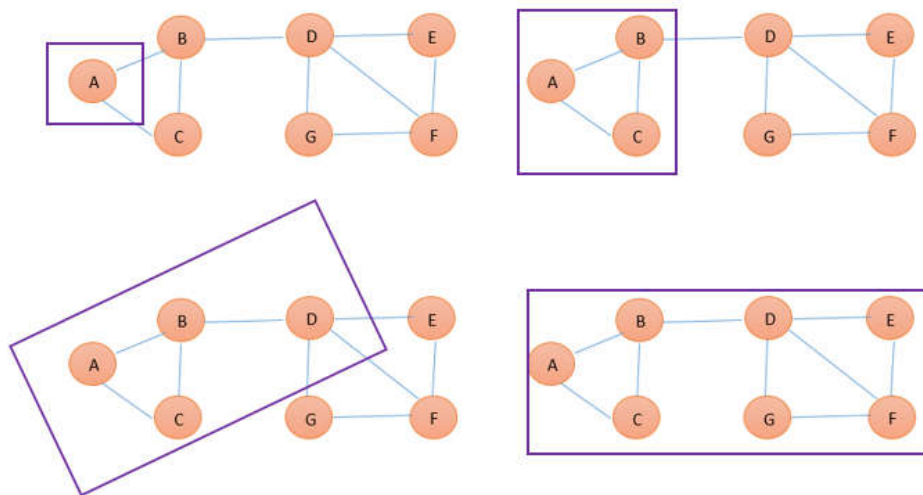
Both execution results (visited nodes respectively) are given in above figure. In each cases all of the 5 nodes are visited so that we can say that the graph is strongly connected.

B. REPORT

1. Theoretical Details

Shortest Path

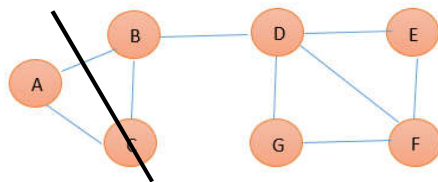
We can use Dijkstra's algorithm to find the shortest path from node s to t . Here (in part a) all edges has weight as 1 and its undirected graph so that we can apply the algorithm. It is a greedy algorithm and continues iteratively. First, it takes the source node as initial point then finds its neighbors (in our scenario all weights are equal). If the node is not neighbors of node s its distance is taken as infinite. The closest node that has lower weight is selected. Node s and the selected neighbor are taken into same set and the process is continues by exploring the closest node to the nodes in the set. Goal is covering all nodes in shortest path not the edges.



Betweenness of the graph

In order to find betweenness of the graph, we can follow an algorithm as follows:

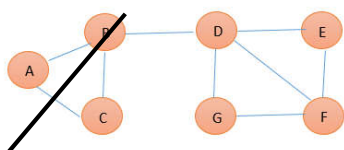
For Edge A-B



We can analyze the graph into 2 subgraph and also we should find the number of reachable nodes for each subgraph as follows:

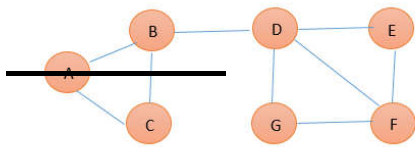
Subgraph1: [A]	→ 1	} 1x5=5
Subgraph2: [B D E F G]	→ 5	

For Edge A-C



Subgraph1: [A]	→ 1	} 1x1=1
Subgraph2: [C]	→ 1	

For Edge B-C



Subgraph1: [C]

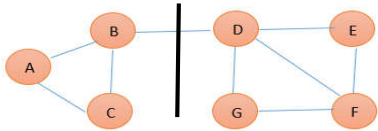
→1

Subgraph2: [B D E F G]

→5

$$1 \times 5 = 5$$

For Edge B-D



Subgraph1: [A B C]

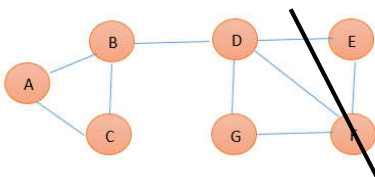
→3

Subgraph2: [D E F G]

→4

$$3 \times 4 = 12$$

For Edge D-E



Subgraph1: [E]

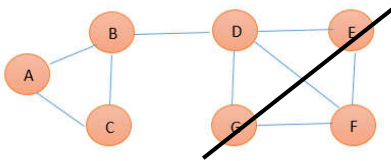
→1

Subgraph2: [A B C D G]

→5

$$1 \times 5 = 5$$

For Edge D-F



Subgraph1: [F]

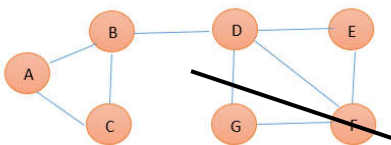
→1

Subgraph2: [A B C D]

→4

$$1 \times 4 = 4$$

For Edge D-G



Subgraph1: [G]

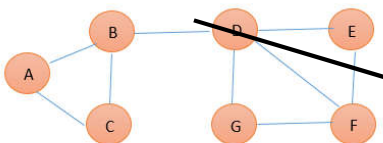
→1

Subgraph2: [A B C D E]

→5

$$1 \times 5 = 5$$

For Edge E-F



Subgraph1: [E]

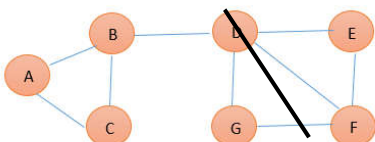
→1

Subgraph2: [F G]

→2

$$1 \times 2 = 2$$

For Edge G-F



Subgraph1: [E F]

→2

Subgraph2: [G]

→1

$$1 \times 2 = 2$$

Strong Connectivity

Strong connectivity is defined as

- Select any node from the graph
- Run BFS starting from this node
- Run BFS starting from this node
(At this time, graph should be reversed. Due to being directed graph, reverse means that change the edges to the opposite direction.)
- If both executions of the algorithm results same number of nodes of the graph it means that we can reach all possible nodes from the selected node and vice a versa.

2. Implementation Details

Pseudo code and complexity Analysis for each part

We build graph class for both part a, part b and part c as can be seen below.

```
class graph
{
    int nodecounter; // Num of nodes in graph
    list<int> *adjacency; // Pointer to an array containing adjacency lists
public:
    graph(int nodecounter);
    void addEdge(int u, int v); //edge between u and v
    void deleteEdge(int u, int v);
    void printpath(int , int , bool [], int [], int &);
};
```

Graph class is defined to build a graph structure from adjacency matrix. Here,

Attributes:

Nodecounter → Number of nodes in the graph (7 in our case)

Adjacency → Pointer to C++ List in order to reach adjacency list of graph

Functions:

Graph(int nodecounter) → Constructor of the graph class

void addEdge (u,v) → It adds an edge from u to v (v to list of u).

void deleteEdge (u,v) → It deletes an edge from u to v (v from list of u).

void printpath(..) → prints all the possible paths from s to d

Key function of the code is this one so that it is important to understand how it works.

printpath (s, d, visited, path, pi)

```
//Initialization phase  
Visited[i] =false for each node  
int *path = new int[linenum]  
int pi=0;//holds path index  
path[pi++]←s  
if (s==d) //if we are at the destination node  
    print related nodes up to this node  
else  
    pass over the adjacency matrix by using i index  
    if it is not visited call resursively printpath(*i, d, visited, path, pi)  
    pi--;  
visited[s] = false;
```

Part a

In addition to graph structure we have defined path structure named as graphpath for this part.

```
struct graphpat{  
    public:  
        int pno;  
        int nodes[20];  
        int length;  
        void addNode(int pno,int node);  
        int findlength(int pno);  
};
```

Attributes:

- | | |
|---------------|---|
| Pno | → Number of the path
(Paths are hold in a global array and we can reach them by using pno) |
| Nodes | → It holds the list of nodes of the path |
| Length | → It holds the length of the path |

Functions:

`void addNode(int);` → It adds node (2nd parameter) to the path that specified by pno

`int findlength(int pno);` → It returns the length of the specified path with pno.

Methodology for part a:

- 1) Read from the file input1.txt and create matrix named adjmat
- 2) Print adjacency matrix of the graph to screen
- 3) Create graph object from graph class and build graph using addEdge function
- 4) Call printhpath function for specified nodes as source and destination
- 5) Create array of paths named mypaths
- 6) Fill mypaths array with paths found in printpath (s,d) function
- 7) Use findlength(m) function to obtain lengths of the all paths from s to d

Complexity for part a

Assume line number is n

	Line in code	Complexity
Step 1 in methodology	Line 137, while loop	O(n)
	Line 146, Nested while loop	O(n ²)
Step 2 in methodology	Line 159, Nested for loop	O(n ²)
Step 3 in methodology	Line 170, Nested for loop	O(n ²)
Step 4 in methodology	Line 188, Nested for loop	O(n)
	Line 191, call for printpath	
Step 5 and 6 in methodology	Line 195, Nested while loop	O(m+n) Where m is the path count
Step 7 in methodology	Line 216, while loop	O(m)

Part b

Note: I have did this part after implementation of part c so that I used BFS in part b.

Methodology for part b:

(Assume we have built graph by using adjacency that read from the file)

- 1) Built new adjacency matrix which supplies following conditions:
 - Delete edge between node1 and node2(entered by the user)
 - If there exist any node that has edge with both node1 and node2, delete this node
And cut its edges with other nodes.
- 2) Create new graph from newly built adjacency matrix.
- 3) Run BFS for both node1 and node2 over the newly created graph so that we can obtain how many nodes are reachable from node1 and node2 separately.
- 4) Print the betweenness value for a (node1,node) edge

In order to implement Step 1 in above methodology we have did the followings in $O(n^2)$ complexity.

```
107 for(i=0;i<linenum;i++) {
108     for(k=0;k<linenum;k++) {
109         if( (i==s && k==d) || (i==d && k==s) )
110             adjmat1[i][k]=0;
111         else
112             adjmat1[i][k]=adjmat[i][k];
113     }
114 }
115
116 for(k=0;k<linenum;k++) {
117     if( (adjmat1[s][k]==1) && (adjmat1[d][k]==1) )
118     {for(i=0;i<linenum;i++)
119         adjmat1[k][i]=0;
120         for(i=0;i<linenum;i++)
121             adjmat1[i][k]=0;}
122     }
123 }
```

Complexity for part b

	Line in code	Complexity
Step 1 in methodology	Line 107, Nested for loop	$O(n^2)$
Step 2 in methodology	Line 124, Nested for loop	$O(n^2)$
Step 3 in methodology	Line 131 and 132, call for BFS function	$O(m+n) + O(m+n)$
Step 4 in methodology	Line 133	$O(1)$

Part c

In addition to graph structure we have defined edge structure for this part.

```
struct edge{  
    public:  
        int node1;  
        int node2;  
        int degr;  
        bool isexist(int n1,int n2);  
};
```

Attributes:

Node1, Node2 → It represents the nodes u and v for edge between u and v.

degr → It is designed to hold betweenness value for this edge

Functions:

bool isexist(int n1, int n2); → Checks whether there is an edge between n1 and n2 or not.

BFS:

We have implemented BFS over the structure as stated in Recitation2 page 5/44. Pseudo code is given below for BFS.

Note that we have implemented the algorithm using queue structure of the C++.

```
int graph::BFS(int start)
```

Visited[i] = false for each node except for the source node

list<int> BFSqueue; **//Initialize empty queue**

BFSqueue.push_back(start);

while(!BFSqueue.empty()) **//if queue is empty it should finish**

 BFSqueue.pop_front();

 For each node(i) adjacent to start node

 If i is not visited

 Mark i as visited and BFSqueue.push_back(i);

Methodology for part c:

- 1) Read from the file `input2.txt` and create matrix named `adjmat`
- 2) Print adjacency matrix of the graph to screen
- 3) Create graph object from graph class and build graph using `addEdge` function
- 4) Create new graph object named `grev` by reversing directions
- 5) Run BFS from any node over graph `g`
- 6) Run BFS from the same node over the graph `grev`
- 7) Count how many nodes are visited during both executions
- 8) If they are equal to the number of nodes in the graph print to screen as it is strongly connected graph otherwise print as not.

Complexity for part c

Assume line number is `n`

	Line in code	Complexity
Step 1 in methodology	Line 150, while loop	$O(n)$
	Line 159, Nested while loop	$O(n^2)$
Step 2 in methodology	Line 172, Nested for loop	$O(n^2)$
Step 3 in methodology	Line 183, Nested for loop	$O(n^2)$
Step 4 in methodology	Line 195, Nested for loop	$O(n^2)$
Step 5 and 6 in methodology	Line 212 and 215, call for BFS function	$O(m+n) + O(m+n)$
Step 7 in methodology	Line 217, if and print statements	$O(1)$