**CHAPTER THREE**

**3.0 METHODOLOGY**

**3.1 Introduction**

A computer program is simply an implementation of an algorithm on a computer. More often, there may be a question arises as to which algorithm should be used. Is one algorithm better than another? This is a nontrivial question which leads to the analysis of algorithms and the means by which they can be compared. The absolute growth depends on the machine used to execute the program, the compiler used to construct the program and what are they?

A programmer, usually as a choice of data structures and algorithms to use. Choosing the best one for a particular job involving, among other factors, two important measures:

***Time complexity*:** How much time will the program take?

***Space complexity*:** How much storage will the program need?

A programmer will sometimes seek a tradeoff between space and time complexity. For example, a programmer might choose a data structure that requires a lot of storage in order to reduce the computation time. To compare the efficiency of algorithms, Juris Hartmains and Richard Steams developed a computational complexity which is a measure of the degree of the difficulty of an algorithm (Alfred et al, 1974).

**3.2 APPROACH**

The following approaches were used:

(1) Extensive study of Dijkstra, Floyd-Warshall, and Bellman Ford Shortest Path Algorithms was carried out.

(2) The use of C#, C++, Java, and Visual BASIC programming languages to code the selected shortest path algorithms.

(3) Evaluation of complexity of the shortest path algorithms using Requirement Based Complexity (RBC) measure.

(4) Analysis and comparative study of the complexities of these set of shortest path algorithms.

**3.3 Empirical analysis and comparism of Implemented Dijkstra algorithm using Requirement Based Complexity (RBC) Measures.**

Complexity of Dijkstra algorithm is the amount of resources necessary to execute it. The following are the RBC values of Dijkstra algorithm when implemented in C#, C++, Java and Visual BASIC programming languages, and tested with the graph shown in figure 3.1:



Fig. 3.1.A diagrammatic representation of a typicalGraph G (V, E)

**3.3.1 Evaluation of Complexity of C# Language Implementation of Dijkstra Shortest path Algorithm**

/\* A C# Program to compute the Shortest Path using Dijkstra Algorithm

\* This is a Class name: DijkstraShortestPath.cs

\* Project Developed by:

\*/

using System;

namespace DijkstraShortestPathProject

{

class DijkstraShortestPath

{

private const int MAXNODE = 10;

private const int TRUE = 1;

private const int FALSE = 0;

private const int INFINTY = 9999;

private const int VISITED = 1;

private const int UNVISITED = 0;

private int[ , ] Weight;

public int noOfNodes;

private char[] Node;

public DijkstraShortestPath(int N)

{

setNoOfNodes(N);

Weight = new int[N,N];

Node = new char[N];

}

private int getNoOfNodes(){

return noOfNodes;

}

private void setNoOfNodes(int Nnodes){

noOfNodes = Nnodes;

}

public void ShortestPath()

{

int ch;

if (noOfNodes <= 0 || noOfNodes > MAXNODE)

{

Console.WriteLine("Wrong Input! Please, Read in correct digit from 1 to 10:");

}

else

{

ReadNodes(Node); // Read in N nodes

ReadWeight(Weight, Node); //Read in weight/cost of each edges

DisplayWeight(Weight); // Display Matrix of Weight

Console.WriteLine("\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ");

Console.WriteLine("Press 1 -> Yes \n Press 0 or any other digit -> No\n");

ch = Convert.ToInt32(Console.ReadLine());

if(ch == 1)

{

Console.Write("\nRead in the Source Node/Vertex: ");

char source = Convert.ToChar(Console.ReadLine());

Console.Write("\nRead in the Target Node/Vertex: ");

char target = Convert.ToChar(Console.ReadLine());

FindShortestPath(Node, Weight, source, target);

}

}

}

private void ReadNodes(char[] Nodes)

{

int i;

Console.WriteLine("Read in the names of all Nodes/Vertices in the graph");

for(i = 0; i < Nodes.Length; i++)

{

Console.Write("\nRead in the names of Node/Vertex " + (i+1) + ":\t" );

Nodes[i] = Convert.ToChar(Console.ReadLine());

}

}

private void ReadWeight(int[,] W, char[] Nodes)

{

int i, j;

Console.Write("\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n");

for(i = 0; i < W.GetLength(0); i++)

{

Console.WriteLine();

for (j = 0; j < W.GetLength(1); j++)

{

Console.Write("Read in the weight/cost of edge between Node "+ Nodes[i]

+ " and Node " + Nodes[j] + ": ");

W[i,j] = Convert.ToInt32(Console.ReadLine());

if (W[i,j] == 0) W[i,j] = INFINTY;

}

}

}

private void DisplayWeight(int[,] W)

{

int i, j;

Console.Write("\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n");

for (i = 0; i < W.GetLength(0); i++)

{

for (j = 0; j < W.GetLength(1); j++)

{

if(W[i,j] == INFINTY) Console.Write(" INF");

else

Console.Write("{0,6}", W[i,j]);

}

Console.WriteLine();

}

}

private void FindShortestPath(char[] Nodes, int[,] length, char source, char target)

{

int souce = 0, j, u, v, taget = 0, alt, undefined = -1;

int[] dist, prev, Q, visited, S;

int found = FALSE;

/\* Dynamically Allocate Array \*/

dist = new int[Nodes.Length];

prev = new int[Nodes.Length];

Q = new int[Nodes.Length];

visited = new int[Nodes.Length];

S = new int[Nodes.Length];

/\* Verify source \*/

for (j = 0; j < Nodes.Length; j++)

{

if(source == Nodes[j])

{

souce = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ Console.Write("\nWrong Input! Source Node not found; No path found: "); return;}

found = FALSE;

/\* Verify target \*/

for(j = 0; j < Nodes.Length; j++)

{

if(target == Nodes[j])

{

taget = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ Console.Write("\nWrong Input! Target Node not found; No path found: "); return; }

dist[souce] = 0; //distance from source to source

prev[souce] = undefined; //previous node in optimal path initialization

for(v = 0; v < Nodes.Length; v++)

{

if(v != souce)

{

dist[v] = INFINTY; //unknown distance function from source to v

prev[v] = undefined; //previous node in optimal path from source

}

Q[v] = v;

visited[v] = UNVISITED;

}

u = Q[minDist(dist, visited)]; //vertex u in Q with min dist[u]

visited[u] = VISITED;

while (visited[taget] == UNVISITED) //While (Q is not empty)

{

for(v = 0; v < Nodes.Length; v++)

{

alt = dist[u] + length[u,v];

if((alt < dist[v]) && (visited[v] == UNVISITED))

{

dist[v] = alt;

prev[v] = u;

}

}

u = Q[minDist(dist, visited)]; //vertex u in Q with min dist[u]

visited[u] = VISITED;

}

//return dist[], prev[]

Console.Write("\nDistance table: \n");

for(v = 0; v < Nodes.Length; v++)

{

if(dist[v] == INFINTY) Console.Write("INF\t");

else

Console.Write(dist[v] + " \t");

}

Console.Write("\nPrev table: \n");

for(v = 0; v < Nodes.Length; v++) Console.Write(prev[v]+1 + " \t");

//getting the shortest path

j = 0;

u = taget;

while(prev[u] != undefined) //construct the shortest path with a stack S

{

S[j++] = u; //Push the vertex onto the stack

u = prev[u]; //Traverse from target to source

}

//display shortest path

Console.Write("\n\nSuccessful, A Path Found!!! \nThe Shortest Path between " +

source + " and " + target + " is:\n " + source + "--->");

for(v = j-1; v > 0; v--) Console.Write(Nodes[S[v]]+ "--->");

Console.Write(Nodes[S[0]]+ "\n");

Console.Write("\nThe weight/cost of the shortest path between Node: " +

source + " and Node: " + target + " is " + dist[taget] + "\n");

}

private int minDist(int[] Dist, int[] visit)

{

int v, temp = INFINTY, n = -1;

for(v = 0; v < Dist.Length; v++)

{

if((temp > Dist[v])&& (visit[v] == UNVISITED)) {temp = Dist[v]; n = v; }

}

return n;

}

}

}

/\* A C# Program to compute the Shortest Path using Dijkstra Algorithm

\* This is a main Class name: DijkstraShortestPathMain.cs

\* Project Developed by:

\*/

using System;

namespace DijkstraShortestPathProject

{

class DijkstraShortestPathMain

{

static void Main(string[] args)

{

int n;

Console.WriteLine("Welcome to C# Program developed to implement Dijkstra Algorithm:");

Console.WriteLine("=================================================================\n");

Console.Write("Read in Number of Vertices/Nodes in Graph: ");

n = Convert.ToInt32(Console.ReadLine());

DijkstraShortestPath dijkstra = new DijkstraShortestPath(n);

dijkstra.ShortestPath();

}

}

}

**RBC measure on Dijkstra Algorithm implementation with C# Language**

**Input**

Extracting from the graph in fig. 3.1 above:

N = Number of vertices = 6 ⇒ Size (n) = 1

V = Vertices defined as {V1, V2, …, Vn} = { 1, 2, 3, 4, 5, 6 } ⇒ Size (V) = 6

M = Number of Edges = 9 ⇒ Size (M) = 1

E = Edges defined as {Weight (u, v): u, v E V} = { 7, 9,10, 14, 2, 11, 15, 9, 6}

= Size (E) = 9

A = Source vertex = 1, ⇒ size (a) = 1

B = Target vertex = 5, ⇒ size (b) = 1

Dist = Distance = {0, ∞, ∞, ∞, ∞, ∞}, ⇒ Size (Dist) = 6

Number of Input = Size (n) + Size(V) + Size (m) + Size (E) + Size ( a ) + Size (b) + Size (Dist) = 1 + 6 + 1 + 9 + 1 + 1 + 6 = 25

**Output**

The shortest path form the source a to the target b is given as

Prev = { 1, 3, 6, 5}, size (prev) = 4

The list of the shortest path, which is given as

C = 9 + 2 + 9 = 20, size (c ) = 1

Number of output = Size (prev) + Size (c ) = 4 + 1 = 5

Number of Interface = 1 (console windows)

Number of files = 1 ( for data storage)

**1. Input Output Complexity (IOC)**

IOC = No of Input + No of Output + No of interfaces + No of files

= 25 + 5 + 1 + 1 = 32

**2. Functional Requirement (FR)**

No of function ⇒ (DijkstraShortestPath ( ), SetNoOfNodes ( ), GetNoOfNodes ( ), ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), FindShortestPath ( ), minDist ( ) ) = 9

No of sub-process/sub-functions ⇒ Every function Fi has zero, one or more sub- process such as Arithmetic, Computation, Display, etc.

Function 1 - ShortestPath ( ) has 3 (Display operations, Read operations, Type-Casting operations)

Function 2 - ReadNodes ( ) has 3 (Display operations, Read operations, Type-Casting operations)

Function 3 - ReadWeight ( ) has 3 (Display, Read, Type-casting operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - FindShortestPath ( ) has 2 (Display, arithmetic addition)

= 3 + 3 + 3 + 1 + 2 = 12

FR = Number of functions \* = 9 \* 12 = 108

**3. Non functional Requirement (NFR)**

Number of Non-FR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 108 + 0 = 108

**5. Product Complexity (PC)**

PC = IOC \* RC = 32 \* 108 = 3456

**6**. **Personal Complexity Attributes (PCA)**

PCA = 1.17 (Suppose Programmer Capability = Low)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((3456 \* 1.17) + 0 + 0 + 0 ) \* 1 = 4043.52

**3.3.2 Evaluation of Complexity of C++ Language Implementation of Dijkstra Shortest path Algorithm**

/\* A C++ Program to compute the Shortest Path using Dijkstra Algorithm

\* Project Developed by

\*/

#include <cstdlib>

#include <iostream>

#include <conio.h>

#include <stdlib.h>

#include <ctype.h>

#include <math.h>

#define MAXNODE 10

#define TRUE 1

#define FALSE 0

#define INFINTY 9999

#define VISITED 1

#define UNVISITED 0

using std::cout;

using std::cin;

class Dijkstra {

private:

int Weight[MAXNODE][MAXNODE], n, ch;

char Node[MAXNODE];

public:

void ReadNodes(char Nodes[], int N);

void ReadWeight(int W[][MAXNODE], int N, char Nodes[]);

void DisplayWeight(int W[][MAXNODE], int N);

void DijkstraShortestPath(char Nodes[], int Length[][MAXNODE], int M, char source, char target);

int minDist(int Dist[], int M, int visit[]);

void ShortestPath();

};

int main()

{

Dijkstra dijkstra;

dijkstra.ShortestPath();

system("PAUSE");

return 0;

}

void Dijkstra::ShortestPath()

{

char source, target;

cout<<"Welcome to C++ Program developed to implement Dijkstra Algorithm:";

cout<<"\n================================================================\n";

cout<<"Read in Number of Vertices/Nodes in Graph: ";

cin>> n;

if (n <= 0 || n > MAXNODE)

{

cout<<"\nWrong Input! Please, Read in correct digit from 1 to 10:";

}

else

{

ReadNodes(Node, n); // Read in N nodes

ReadWeight(Weight, n, Node); //Read in weight/cost of each edges

DisplayWeight(Weight, n); // Display Matrix of Weight

cout<<"\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ";

cout<<"Press 1 -> Yes \n Press 0 or any other digit -> No\n";

cin>>ch;

if(ch == 1)

{ cout<<"\nRead in the Source Node/Vertex: "; cin>>source;

cout<<"\nRead in the Target Node/Vertex: "; cin>>target;

DijkstraShortestPath(Node, Weight, n, source, target);

} //end if

} //end else

}

void Dijkstra::ReadNodes(char Nodes[], int N)

{

int i;

cout<<"Read in the names of all Nodes/Vertices in the graph\n";

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

{

cout<<"\nRead in the names of Node/Vertex "<<i+1<<":\t";

//char c = getch(); cout"%c", c);

cin>>Nodes[i];

}

Nodes[i] = '\0';

}

void Dijkstra::ReadWeight(int W[][MAXNODE], int N, char Nodes[])

{

int i, j;

cout<<"\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n";

for(i = 0; i < N; i++) {

cout<<"\n";

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

{

cout<<"Read in the weight/cost of edge between Node "<<Nodes[i]<<" and Node "<<Nodes[j]<<": ";

cin>>W[i][j];

if (W[i][j] == 0) W[i][j] = INFINTY;

}

}

}

void Dijkstra::DisplayWeight(int W[][MAXNODE], int N)

{

int i, j;

cout<<"\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n";

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

{

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

{

if(W[i][j] == INFINTY) cout<<"INF\t";

else

cout<<W[i][j]<<"\t";

}

cout<<"\n";

}

}

void Dijkstra::DijkstraShortestPath(char Nodes[], int Length[][MAXNODE], int M, char source, char target)

{

int souce, j, alt, u, v, taget;

int \*dist, \*prev, \*Q, \*visited, \*S;

int found = FALSE, undefined = -1;

/\* Dynamically Allocate Array \*/

dist = (int \*) (malloc(M \* sizeof(int)));

prev = (int \*) (malloc(M \* sizeof(int)));

Q = (int \*) (malloc(M \* sizeof(int)));

visited = (int \*) (malloc(M \* sizeof(int)));

S = (int \*) (malloc(M \* sizeof(int)));

/\* Verify source \*/

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

{

if(source == Nodes[j])

{

souce = j; //get the index position of source from the Node

found = TRUE;

break;

}

}

if (found == FALSE)

{ cout<<"\nWrong Input! Source Node not found; No path found: "; exit(0);}

found = FALSE;

/\* Verify target \*/

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

{

if(target == Nodes[j])

{

taget = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ cout<<"\nWrong Input! Target Node not found; No path found: "; exit(0); }

dist[souce] = 0; //distance from source to source

prev[souce] = undefined; //previous node in optimal path initialization

for(v = 0; v < M; v++)

{

if(v != souce)

{

dist[v] = INFINTY; //unknown distance function from source to v

prev[v] = undefined; //previous node in optimal path from source

}

Q[v] = v;

visited[v] = UNVISITED;

}

u = Q[minDist(dist, M, visited)]; //vertex u in Q with min dist[u]

visited[u] = VISITED;

while (visited[taget] == UNVISITED) //While (Q is not empty)

{

for(v = 0; v < M; v++)

{

alt = dist[u] + Length[u][v];

if((alt < dist[v]) && (visited[v] == UNVISITED))

{

dist[v] = alt;

prev[v] = u;

}

}

u = Q[minDist(dist, M, visited)]; //vertex u in Q with min dist[u]

visited[u] = VISITED;

}

//return dist[], prev[]

cout << "\nDistance table: \n" ;

for(v = 0; v < M; v++)

{

if(dist[v] == INFINTY) cout<<"INF\t";

else

cout<<dist[v]<<" \t";

}

cout<<"\nPrev table: \n";

for(v = 0; v < M; v++) cout<<prev[v]+1<<" \t";

//getting the shortest path

j = 0;

u = taget;

while(prev[u] != undefined) //construct the shortest path with a stack S

{

S[j++] = u; //Push the vertex onto the stack

u = prev[u]; //Traverse from target to source

}

//display shortest path

cout<<"\n\nSuccessful, A Path Found!!! \nThe Shortest Path between "

<<source<<" and "<<target<<" is: "<<"\n "<<source<<"--->";

for(v = j-1; v > 0; v--) cout<<Node[S[v]]<<"--->";

cout<<Node[S[0]]<<"\n";

cout<<"The weight/cost of the shortest path between Node: "<<source<<" and Node: "

<<target<<" is: "<<dist[taget]<<"\n";

}

int Dijkstra::minDist(int Dist[], int M, int visit[])

{

int v, temp = INFINTY, n = -1;

for(v = 0; v < M; v++)

{

if((temp > Dist[v])&& (visit[v] == UNVISITED)) {temp = Dist[v]; n = v; }

}

return n;

}

**RBC measure on Dijkstra Algorithm implementation with C++ Language**

**1. Input Output Complexity (IOC)**

Number of Input = Size (n) + Size(V) + Size (m) + Size (E) + Size ( a ) + Size (b) + Size (Dist) = 1 + 6 + 1 + 9 + 1 + 1 + 6 = 25

Number of output = Size (prev) + Size (c) = 4 + 1 = 5

Number of Interface = 1 (console window)

Number of files = 1 (for data storage)

IOC = No of Input + No of Output + No of interfaces + No of files

= 25 + 5 + 1 + 1 = 32

**2. Functional Requirement (FR)**

No of functions ⇒ (DijkstraShortestPath ( ), ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), minDist ( )) = 6

No of sub-process/sub-functions ⇒ Every function Fi has zero, one or more sub-process such as Arithmetic, Computation, Display, etc.

Function 1 - ShortestPath ( ) has 2 (Display operations, Read operations )

Function 2 - ReadNodes ( ) has 2 (Display operations, Read operations)

Function 3 - ReadWeight ( ) has 2 (Display operations, Read operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - DijkstraShortestPath ( ) has 2 (Display, arithmetic addition)

= 2 + 2 + 2 + 1 + 2 = 9

FR = Number of functions \* = 6 \* 9 = 54

**3. Non functional Requirement (NFR)**

Number of Non-FR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 54 + 0 = 54

**5. Product Complexity (PC)**

PC = IOC \* RC = 32 \* 54 = 1728

**6**. **Personal Complexity Attributes (PCA)**

PCA = 0.90 (Suppose Programmer Capability = High)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((1728 \* 0.90) + 0 + 0 + 0 ) \* 1 = 1555.2

**3.3.3 Evaluation of Complexity of VB 6.0 Language Implementation of Dijkstra Shortest path Algorithm**

' A VB 6.0 -PROGRAMMING LANGUAGE IMPLEMENTATION OF DIJKSTRA SHORTEST PATH ALGORITHM

Const MAXNODE = 10

Const INFINTY = 9999

Const VISITED = 1

Const UNVISITED = 0

Private Function minDist(ByRef Dist() As Integer, ByRef visit() As Integer, ByVal M As Integer) As Integer

Dim v As Integer, temp As Integer, n As Integer

temp = INFINTY: n = -1

For v = 1 To M

If ((temp > Dist(v)) And (visit(v) = UNVISITED)) Then

temp = Dist(v): n = v

End If

Next v

minDist = n

End Function

'\*Procedure for finding the shortest path \*

Private Sub FindShortestPath(ByRef Nodes() As String, ByRef Length() As Integer, ByVal source As String, ByVal target As String, ByVal M As Integer)

Dim souce As Integer, j As Integer, u As Integer, v As Integer, taget As Integer, alt As Integer

Const undefined As Integer = -1

Dim Dist() As Integer, prev() As Integer, Q() As Integer, visit() As Integer, S() As Integer

Dim found As Boolean

found = False

'\* Dynamically Allocate Array \*

ReDim Dist(M)

ReDim prev(M)

ReDim Q(M)

ReDim visit(M)

ReDim S(M)

'\* Verify source \*

For j = 1 To M

If (source = Nodes(j)) Then

souce = j

found = True

'break

End If

Next j

If (found = False) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Source Node not found; No path found: "

Return

End If

found = False

'\* Verify target \*

For j = 1 To M

If (target = Nodes(j)) Then

taget = j

found = True

'break

End If

Next j

If (found = False) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Target Node not found; No path found: "

Return

End If

Dist(souce) = 0 'distance from source to source

prev(souce) = undefined 'previous node in optimal path initialization

For v = 1 To M

If (v <> souce) Then

Dist(v) = INFINTY 'unknown distance function from source to v

prev(v) = undefined 'previous node in optimal path from source

End If

Q(v) = v

visit(v) = UNVISITED

Next v

u = Q(minDist(Dist, visit, M)) '//vertex u in Q with min dist[u]

visit(u) = VISITED

While (visit(taget) = UNVISITED) '//While (Q is not empty)

For v = 1 To M

alt = Dist(u) + Length(u, v)

If ((alt < Dist(v)) And (visit(v) = UNVISITED)) Then

Dist(v) = alt

prev(v) = u

End If

Next v

u = Q(minDist(Dist, visit, M)) ' //vertex u in Q with min dist[u]

visit(u) = VISITED

Wend

'//return dist[], prev[]

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Distance table: " & vbCrLf

For v = 1 To M

If (Dist(v) = INFINTY) Then

TxtOutput.Text = TxtOutput.Text & "INF "

Else

TxtOutput.Text = TxtOutput.Text & Dist(v) & " "

End If

Next v

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Prev table: " & vbCrLf

For v = 1 To M

TxtOutput.Text = TxtOutput.Text & prev(v) & " "

Next v

'//getting the shortest path

j = 1

u = taget

While (prev(u) <> undefined) '//construct the shortest path with a stack S

S(j) = u: j = j + 1 ' //Push the vertex onto the stack

u = prev(u) ' //Traverse from target to source

Wend

' //display shortest path

TxtOutput.Text = TxtOutput.Text & vbCrLf & vbCrLf & "Successful, A Path Found!!! \nThe Shortest Path between " + source + " and " + target + " is: " & vbCrLf + source + "--->"

For v = j - 1 To 2 Step -1

TxtOutput.Text = TxtOutput.Text & Nodes(S(v)) & "--->"

Next v

TxtOutput.Text = TxtOutput.Text & Nodes(S(1))

TxtOutput.Text = TxtOutput.Text & vbCrLf & "The weight/cost of the shortest path between Node: " & source & " and Node: " & target & " is " & Dist(taget)

End Sub

'\*Procedure for reading Nodes \*

Private Sub readNodes(ByRef Nodes() As String, ByVal M As Integer)

Dim j As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the names of all Nodes/Vertices in the graph"

MsgBox "Read in the names of all Nodes/Vertices in the graph", vbOKOnly, "DIJKSTRA"

ReDim Nodes(M)

For j = 1 To M

Nodes(j) = InputBox("Read in the names of Node/Vertex " & j & " : ", "Dijkstra")

Nodes(j) = Left$(Nodes(j), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the names of Node/Vertex " & j & " : " & Nodes(j)

Next j

End Sub

'\*Procedure for reading weight \*

Private Sub readWeight(ByRef W() As Integer, ByRef Nodes() As String, ByVal n As Integer)

Dim j As Integer, i As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the weights/costs of all edges between 2 Vertices in the graph"

MsgBox "Read in the weights/costs of all edges between 2 Vertices in the graph:", vbOKOnly, "DIJKSTRA"

For i = 1 To n

For j = 1 To n

W(i, j) = Val(InputBox("Read in the weight/cost of edge between Node " & Nodes(i) & " and Node " & Nodes(j) & ": ", "Dijkstra"))

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the weight/cost of edge between Node " & Nodes(i) & " and Node " & Nodes(j) & ": " & W(i, j)

If (W(i, j) = 0) Then W(i, j) = INFINTY

Next j

Next i

End Sub

Private Sub DisplayWeight(ByRef W() As Integer, ByVal n As Integer)

Dim i As Integer, j As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "The weights/costs Table of all edges between 2 Vertices in the graph: "

For i = 1 To n

TxtOutput.Text = TxtOutput.Text & vbCrLf

For j = 1 To n

If (W(i, j) = INFINTY) Then

TxtOutput.Text = TxtOutput.Text & "INF "

Else

TxtOutput.Text = TxtOutput.Text & W(i, j) & " "

End If

Next j

Next i

End Sub

Private Sub CmdCancel\_Click()

End

End Sub

Private Sub CmdReset\_Click()

TxtOutput.Text = ""

CmdStart.Enabled = True

End Sub

Private Sub CmdStart\_Click()

'\*Begin the main procedure for implementing Dijkstra Algorithm \*

Dim Node() As String, Weight(MAXNODE, MAXNODE) As Integer

Dim noOfNodes As Integer, Ch As Integer, source As String \* 1, target As String \* 1

TxtOutput.Text = TxtOutput.Text & "Welcome to VB 6.0 Program developed to implement Dijkstra Algorithm:"

TxtOutput.Text = TxtOutput.Text & vbCrLf & "...................................................................."

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in Number of Vertices/Nodes in Graph: "

noOfNodes = Val(InputBox("Read in Number of Vertices/Nodes in Graph: ", "DIjkstra"))

TxtOutput.Text = TxtOutput.Text + Str(noOfNodes)

If (noOfNodes <= 0 Or noOfNodes > MAXNODE) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Please, Read in correct digit from 1 to 10:"

Else

Call readNodes(Node, noOfNodes)

Call readWeight(Weight, Node, noOfNodes)

Call DisplayWeight(Weight, noOfNodes)

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Do you wish to find Shortest Path between two Vertices/Nodes in the Graph: "

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Press 1 -> Yes"

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Press 0 or any other digit -> No"

Ch = Val(InputBox("Press 1 -> Yes, Press 0 or any other digit -> No", "DIjkstra"))

TxtOutput.Text = TxtOutput.Text & vbCrLf & Str(Ch)

If (Ch = 1) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the Source Node/Vertex: "

source = Left(InputBox("Read in the Source Node/Vertex: ", "DIjkstra"), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & source

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the Target Node/Vertex: "

target = Left(InputBox("Read in the target Node/Vertex: ", "DIjkstra"), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & target

Call FindShortestPath(Node, Weight, source, target, noOfNodes)

End If

End If

TxtOutput.Text = TxtOutput.Text & vbCrLf & "\*\*\*\*\*\* Press any Key to Continue \*\*\*\*\*\*"

CmdStart.Enabled = False

End Sub

**RBC measure on Dijkstra Algorithm implementation with VB 6.0 Language**

**1. Input Output Complexity (IOC)**

Number of Input = Size (n) + Size(V) + Size (m) + Size (E) + Size ( a )

+ Size (b) + Size (Dist) = 1 + 6 + 1 + 9 + 1 + 1 + 6 = 25

Number of output = Size (prev) + Size (c) = 4 + 1 = 5

Number of Interface = 1 (console window)

Number of files = 1 (for data storage)

IOC = No of Input + No of Output + No of interfaces + No of files

= 25 + 5 + 1 + 1 = 32

**2. Functional Requirement (FR)**

No of functions ⇒ (ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), FindShortestPath ( ), minDist ( )) = 5

No of sub-process/sub-functions ⇒ Every function Fi has zero, one or more sub- process such as Arithmetic, Computation, Display, etc.

Function 1 - ReadNodes ( ) has 2 (Display operations, Read operations)

Function 2 - ReadWeight ( ) has 2 (Display operations, Read operations)

Function 3 - DisplayWeight ( ) has 1 (Display operations)

Function 4 - FindShortestPath ( ) has 2 (Display operations, arithmetic addition)

= 2 + 2 + 1 + 2 = 7

FR = Number of functions \* = 5 \* 7 = 35

**3. Non functional Requirement (NFR)**

Number of Non-FR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 35 + 0 = 35

**5. Product Complexity (PC)**

PC = IOC \* RC = 32 \* 35 = 1120

**6**. **Personal Complexity Attributes (PCA)**

PCA = 0.90 (Suppose Programmer Capability = High)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((1120 \* 0.90) + 0 + 0 + 0 ) \* 1 = 1008

**3.3.4 Evaluation of Complexity of JAVA Language Implementation of Dijkstra Shortest path Algorithm**

/\* A JAVA Program to compute the Shortest Path using Dijkstra Algorithm

\* Class Name: Dijkstra.java

\* Project Developed by

\*/

import java.util.Scanner;

public class Dijkstra {

private final int MAXNODE = 10;

private final int TRUE = 1;

private final int FALSE = 0;

private final int INFINTY = 9999;

private final int VISITED = 1;

private final int UNVISITED = 0;

private int[][] Weight;

public int noOfNodes;

private char[] Node;

Scanner scanInput = new Scanner(System.in);

public Dijkstra(int N)

{

setNoOfNodes(N);

Weight = new int[N][N];

Node = new char[N];

}

private int getNoOfNodes(){

return noOfNodes;

}

private void setNoOfNodes(int Nnodes){

noOfNodes = Nnodes;

}

public void ShortestPath()

{

int ch;

if (noOfNodes <= 0 || noOfNodes > MAXNODE)

{

System.out.println("Wrong Input! Please, Read in correct digit from 1 to 10:");

}

else

{

ReadNodes(Node); // Read in N nodes

ReadWeight(Weight, Node); //Read in weight/cost of each edges

DisplayWeight(Weight); // Display Matrix of Weight

System.out.println("\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ");

System.out.println("Press 1 -> Yes \n Press 0 or any other digit -> No\n");

ch = scanInput.nextInt();

if(ch == 1)

{

System.out.print("\nRead in the Source Node/Vertex: ");

String str = scanInput.next();

char source = str.charAt(0);

System.out.print("\nRead in the Target Node/Vertex: ");

str = scanInput.next();

char target = str.charAt(0);

FindShortestPath(Node, Weight, source, target);

}

}

}

private void ReadNodes(char[] Nodes)

{

int i;

System.out.println("Read in the names of all Nodes/Vertices in the graph");

for(i = 0; i < Nodes.length; i++)

{

System.out.print("\nRead in the names of Node/Vertex " + (i+1) + ":\t" );

String Str = scanInput.next();

Nodes[i] = Str.charAt(0);

}

}

private void ReadWeight(int[][] W, char[] Nodes)

{

int i, j;

System.out.print("\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n");

for(i = 0; i < W.length; i++)

{

System.out.println();

for (j = 0; j < W[i].length; j++)

{

System.out.print("Read in the weight/cost of edge between Node "+ Nodes[i] + " and Node " + Nodes[j] + ": ");

W[i][j] = scanInput.nextInt();

if (W[i][j] == 0) W[i][j] = INFINTY;

}

}

}

private void DisplayWeight(int[][] W)

{

int i, j;

System.out.print("\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n");

for (i = 0; i < W.length; i++)

{

for (j = 0; j < W[i].length; j++)

{

if(W[i][j] == INFINTY) System.out.printf(" INF");

else

System.out.printf("%6d", W[i][j]);

}

System.out.print("\n");

}

}

private void FindShortestPath(char Nodes[], int Length[][], char source, char target)

{

int souce = 0, j, u, v, taget = 0, alt, undefined = -1;

int[] dist, prev, Q, visited, S;

int found = FALSE;

/\* Dynamically Allocate Array \*/

dist = new int[Nodes.length];

prev = new int[Nodes.length];

Q = new int[Nodes.length];

visited = new int[Nodes.length];

S = new int[Nodes.length];

/\* Verify source \*/

for (j = 0; j < Nodes.length; j++)

{

if(source == Nodes[j])

{

souce = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ System.out.print("\nWrong Input! Source Node not found; No path found: "); return;}

found = FALSE;

/\* Verify target \*/

for(j = 0; j < Nodes.length; j++)

{

if(target == Nodes[j])

{

taget = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ System.out.print("\nWrong Input! Target Node not found; No path found: "); return; }

dist[souce] = 0; //distance from source to source

prev[souce] = undefined; //previous node in optimal path initialization

for(v = 0; v < Nodes.length; v++)

{

if(v != souce)

{

dist[v] = INFINTY; //unknown distance function from source to v

prev[v] = undefined; //previous node in optimal path from source

} Q[v] = v;

visited[v] = UNVISITED;

}

u = Q[minDist(dist, visited)]; //vertex u in Q with min dist[u]

visited[u] = VISITED;

while (visited[taget] == UNVISITED) //While (Q is not empty)

{

for(v = 0; v < Nodes.length; v++)

{

alt = dist[u] + Length[u][v];

if((alt < dist[v]) && (visited[v] == UNVISITED))

{

dist[v] = alt;

prev[v] = u;

}

}

u = Q[minDist(dist, visited)]; //vertex u in Q with min dist[u]

visited[u] = VISITED;

}

//return dist[], prev[]

System.out.print("\nDistance table: \n");

for(v = 0; v < Nodes.length; v++)

{

if(dist[v] == INFINTY) System.out.print("INF\t");

else

System.out.print(dist[v] + " \t");

}

System.out.print("\nPrev table: \n");

for(v = 0; v < Nodes.length; v++) System.out.print(prev[v]+1 + " \t");

//getting the shortest path

j = 0;

u = taget;

while(prev[u] != undefined) //construct the shortest path with a stack S

{

S[j++] = u; //Push the vertex onto the stack

u = prev[u]; //Traverse from target to source

}

//display shortest path

System.out.print("\n\nSuccessful, A Path Found!!! \nThe Shortest Path between " +

source + " and " + target + " is:\n " + source + "--->");

for(v = j-1; v > 0; v--) System.out.print(Nodes[S[v]]+ "--->");

System.out.print(Nodes[S[0]]+ "\n");

System.out.print("\nThe weight/cost of the shortest path between Node: " +

source + " and Node: " + target + " is " + dist[taget] + "\n");

}

private int minDist(int Dist[], int visit[])

{

int v, temp = INFINTY, n = -1;

for(v = 0; v < Dist.length; v++)

{

if((temp > Dist[v])&& (visit[v] == UNVISITED)) {temp = Dist[v]; n = v; }

}

return n;

}

}

/\* A Java Program to compute the Shortest Path using Dijkstra Algorithm

\* This is a main Class name: DijkstraShortestPath.java

\* Project Developed by

\*/

import java.util.Scanner;

public class DijkstraShortestPath {

public static void main(String[] args)

{

Scanner scanInput = new Scanner(System.in);

int n;

System.out.println("Welcome to Java Program developed to implement Dijkstra Algorithm:");

System.out.println("=================================================================\n");

System.out.print("Read in Number of Vertices/Nodes in Graph: ");

n = scanInput.nextInt();

Dijkstra dijkstra = new Dijkstra(n);

dijkstra.ShortestPath();

}

}

**RBC measure on Dijkstra Algorithm implementation with Java Language**

**1. Input Output Complexity (IOC)**

Number of Input = Size (n) + Size(V) + Size (m) + Size (E) + Size ( a )

+ Size (b) + Size (Dist) = 1 + 6 + 1 + 9 + 1 + 1 + 6 = 25

Number of output = Size (prev) + Size (c ) = 4 + 1 = 5

Number of Interface = 1 (console windows)

Number of files = 1 ( for data storage)

IOC = No of Input + No of Output + No of interfaces + No of files

= 25 + 5 + 1 + 1 = 32

**2. Functional Requirement (FR)**

No of functions ⇒ (DijkstraShortestPath ( ), SetNoOfNodes ( ), GetNoOfNodes ( ), ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), FindShortestPath ( ), minDist ( )) = 9

No of sub-process/sub-functions ⇒ Every function Fi has zero, one or more sub- process such as Arithmetic, Computation, Display, etc.

Function 1 - ShortestPath ( ) has 3 (Display operations, Read operations, Type-Casting operations)

Function 2 - ReadNodes ( ) has 3 (Display operations, Read operations, Type-Casting operations)

Function 3 - ReadWeight ( ) has 3 (Display, Read, Type-casting operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - FindShortestPath ( ) has 2 (Display, arithmetic addition)

= 3 + 3 + 3 + 1 + 2 = 12

FR = Number of functions \* = 9 \* 12 = 108

**3. Non functional Requirement (NFR)**

Number of Non-FR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 108 + 0 = 108

**5. Product Complexity (PC)**

PC = IOC \* RC = 32 \* 108 = 3456

**6**. **Personal Complexity Attributes (PCA)**

PCA = 1.00 (Suppose Programmer Capability = Normal)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1, Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((3456 \* 1.00) + 0 + 0 + 0 ) \* 1 = 3456

**3.4 Empirical analysis and comparism of Implemented Floyd-Warshall algorithm using Requirement Based Complexity (RBC) Measures.**

Complexity of Floyd-Warshall algorithm is the amount of resources necessary to execute it. The following are the values of Floyd-Warshall algorithm when implemented in C#, C++, Java and Visual BASIC programming languages, and tested with the graph shown in figure 3.3:



Fig. 3.2.A diagrammatic representation of a typicalGraph G (V, E)

**3.4.1 Evaluation of Complexity of C# Language Implementation of Floyd-Warshall Shortest path Algorithm**

/\* A C# Program to compute the Shortest Path using Floyd Warshall Algorithm

\* This is a Class name: FloydWarshallShortestPath.cs

\* Project Developed by

\*/

using System;

namespace FloydWarshallShortestPathProject

{

class FloydWarshallShortestPath

{

private const int MAXNODE = 10;

private const int TRUE = 1;

private const int FALSE = 0;

private const int INFINTY = 9999;

private const int NULL = -1;

private int[,] Weight;

private int[,] Dist ;

private int[,] Next ;

public int noOfNodes;

private char[] Node;

public FloydWarshallShortestPath(int N)

{

setNoOfNodes(N);

Weight = new int[N,N];

Dist = new int[N,N];

Next = new int[N,N];

Node = new char[N];

}

private int getNoOfNodes(){

return noOfNodes;

}

private void setNoOfNodes(int Nnodes){

noOfNodes = Nnodes;

}

public void ShortestPath()

{

int ch;

if (noOfNodes <= 0 || noOfNodes > MAXNODE)

{

Console.WriteLine("Wrong Input! Please, Read in correct digit from 1 to 10:");

}

else

{

ReadNodes(Node); // Read in N nodes

ReadWeight(Weight, Node); //Read in weight/cost of each edges

DisplayWeight(Weight); // Display Matrix of Weight

ComputeShortestPath(Weight, Dist, Next);

DisplayShortestPath(Dist);

Console.WriteLine("\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ");

Console.WriteLine("Press 1 -> Yes \n Press 0 or any other digit -> No\n");

ch = Convert.ToInt32(Console.ReadLine());

if(ch == 1) FindShortestPath(Dist, Node, Next);

}

}

private void ReadNodes(char[] Nodes)

{

int i;

Console.WriteLine("Read in the names of all Nodes/Vertices in the graph");

for(i = 0; i < Nodes.Length; i++)

{

Console.Write("\nRead in the names of Node/Vertex " + (i+1) + ":\t" );

Nodes[i] = Convert.ToChar(Console.ReadLine());

}

}

private void ReadWeight(int[,] W, char[] Nodes)

{

int i, j;

Console.Write("\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n");

for(i = 0; i < W.GetLength(0); i++)

{

Console.WriteLine();

for (j = 0; j < W.GetLength(1); j++)

{

Console.Write("Read in the weight/cost of edge between Node "+ Nodes[i]

+ " and Node " + Nodes[j] + ": ");

W[i,j] = Convert.ToInt32(Console.ReadLine());

}

}

}

private void DisplayWeight(int[,] W)

{

int i, j;

Console.Write("\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n");

for (i = 0; i < W.GetLength(0); i++)

{

for (j = 0; j < W.GetLength(1); j++)

{

Console.Write("{0,6}", W[i,j]);

}

Console.WriteLine();

}

}

private void ComputeShortestPath(int[,] edge, int[,] dist, int[,] next)

{

int i, j, k, u, v;

/\* initialize dist \*/

for(u = 0; u < edge.GetLength(0); u++)

{

for(v = 0; v < edge.GetLength(1); v++)

{

if (edge[u,v] == 0) { dist[u,v] = INFINTY; next[u,v] = NULL; }

else

{ dist[u,v] = edge[u,v]; next[u,v] = v; }

}

}

/\* Update dist with standard Floyd-Warshall implementation \*/

for(k = 0; k < dist.GetLength(0); k++)

{

for(i = 0; i < dist.GetLength(1); i++)

{

for(j = 0; j < dist.GetLength(0); j++)

{

if(dist[i,j] > dist[i,k] + dist[k,j])

{

dist[i,j] = dist[i,k] + dist[k,j];

next[i,j] = next[i,k];

}

}

}

}

}

private void DisplayShortestPath(int[,] dist)

{

int i, j;

Console.Write("\nThe Shortest Path Table of all path between 2 Vertices in the graph:\n");

for (i = 0; i < dist.GetLength(0); i++)

{

for (j = 0; j < dist.GetLength(1); j++)

{

if(dist[i,j] == INFINTY) Console.Write(" INF");

else

Console.Write("{0,6}", dist[i,j]);

}

Console.Write("\n");

}

}

private void FindShortestPath(int[,] dist, char[] Nodes, int[,] next)

{

char source, target;

int u = 0, j, v = 0, k = 0, cost;

int[] path = new int[Nodes.Length];

int found;

found = FALSE;

Console.Write("\nRead in the Source Node/Vertex: ");

source = Convert.ToChar(Console.ReadLine());

Console.Write("\nRead in the Target Node/Vertex: ");

target = Convert.ToChar(Console.ReadLine());

/\* Verify source \*/

for (j = 0; j < Nodes.Length; j++)

{

if(source == Nodes[j])

{

u = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ Console.Write("\nWrong Input! Source Node not found; No path found: "); return;}

found = FALSE;

/\* Verify target \*/

for(j = 0; j < Nodes.Length; j++)

{

if(target == Nodes[j])

{

v = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ Console.Write("\nWrong Input! Target Node not found; No path found: "); return; }

cost = dist[u,v]; //get the shortest distance between source and target

//procedure to construct path

if(next[u,v] == NULL)

{ Console.Write("\n Failure, No path found! "); return; }

path[k] = u;

while(u != v)

{

u = next[u,v];

path[++k] = u;

}

//display shortest path

Console.Write("\n\nSuccessful, A Path Found!!! \nThe Shortest Path between " +

source + " and " + target + " is:\n " + source + "--->");

for(j = 1; j < k; j++) Console.Write(Node[path[j]]+ "--->");

Console.Write(Node[path[k]]+ "\n");

Console.Write("\nThe weight/cost of the shortest path between Node: " +

source + " and Node: " + target + " is " + cost + "\n");

}

}

}

**RBC measure on Floyd – Warshall Algorithm implemented with C#**

**Input**

Extracting from the graph in fig. 3.2 above:

N = Number of vertices V = 4, size (N) = 1

V = { V1, V2, V3, V4} = {1, 2, 3, 4} = size (V) = 4

M = Number of Edges = 5, size (M) = 1

E = {w (u, v): u, v € V} = { 4, -2, 2, -1, 3}, size (E) = 5

Dist = |V| x |V| array of minimum distance initialized to (infinity)

Where N = |V| = 4

Size (Dist) = 4 x 4 = 16

Next = |V| \* |V| array of vertex indices initialized to null

Size (next) = 4 x 4 = 16

A = Source vertex = 2, size (a) = 1

B = Target vertex = 3, size (b) = 1

No of input = Size (N) + Size (V) + Size (M) + Size (E) + Size (Dist) + Size (Next) + Size (a) + Size (b) = 1 + 4 + 1 + 5 +16 + 16 + 1 + 1 = 45

**Output**

* Dist = |V| x |V| array of the computed minimum or shortest distance from any

Source vertex you to destination vertex V

* The shortest path constructed from a given source a to target vertex b, which is:

Path = { 2, 1, 3}, Size (path) = 3

* The cost/weight of the shortest path computed as; c = 4 + (-2) = 2, size (c ) = 1

No of output = Size (Dist) + Size (Path) + Size (c ) = 16 + 3 + 1 = 20

No of interface = 1 (console window)

No of files = 1 (Data storage)

1. **Input Output Complexity (IOC)**

IOC = No of input + No of output + No of Interface + No of files

= 45 + 20 + 1 + 1 = 67

1. **Functional Requirement (FR)**

No of functions = (FloydWarshellShortestPath ( ), SetNoOfNodes ( ), GetNoOfNodes ( ), ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), ComputeShortestPath ( ), DisplayShortestPath ( ) FindShortestPath ( ) = 10

**No of sub-process/sub- functions**

Function 1 - ShortestPath ( ) has 3 (Display, Read, Type-Casting

operations)

Function 2 - ReadNodes ( ) has 3 (Display, Read, Type-Casting

operations)

Function 3 - ReadWeight ( ) has 3 (Display, Read, Type-casting operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - ComputeShortestPath( ) has 1 (arithmetic addition)

Function 6 - DisplayShortestPath ( ) 1 (Display operation)

Function 7 - FindShortestPath ( ) 4 (Display, Read, Type-Casting &

increment operations)

= 3 + 3 + 3 + 1 + 1 + 1 + 4 = 16

FR = Number of functions \* = 10 \* 16 = 160

**3. Non functional Requirement (NFR)**

NFR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 160 + 0 = 160

**5. Product Complexity (PC)**

PC = IOC \* RC = 67 \* 160 = 10720

**6. Personal Complexity Attributes (PCA)**

PCA = 1.17 (Suppose Programmer Capability = Low)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((10720 \* 1.17) + 0 + 0 + 0 ) \* 1

= 12542.4

**3.4.2 Evaluation of Complexity of C++ Language Implementation of Floyd-Warshall Shortest path Algorithm**

/\* A C++ Program to compute the Shortest Path using Floyd Warshall Algorithm

\* Project Developed by

\*/

#include <iostream>

#include <conio.h>

#include <stdlib.h>

#include <ctype.h>

#include <math.h>

#define MAXNODE 10

#define TRUE 1

#define FALSE 0

#define Null -1

#define INFINTY 9999

using std::cout;

using std::cin;

class FloydWarshall {

private:

int Weight[MAXNODE][MAXNODE], Dist[MAXNODE][MAXNODE], Next[MAXNODE][MAXNODE], n, ch;

char Node[MAXNODE];

public:

void ReadNodes(char Nodes[], int N);

void ReadWeight(int W[][MAXNODE], int N, char Nodes[]);

void DisplayWeight(int W[][MAXNODE], int N);

void ComputeShortestPath(int W[][MAXNODE], int M, int dist[][MAXNODE], int next[][MAXNODE]);

void DisplayShortestPath(int dist[][MAXNODE], int M);

void FindShortestPath(int dist[][MAXNODE], int M, char Nodes[], int next[][MAXNODE]);

void ShortestPath();

};

int main()

{

FloydWarshall floydWarshall;

floydWarshall.ShortestPath();

system("PAUSE");

return 0;

}

void FloydWarshall::ShortestPath()

{

cout<<"Welcome to C++ Program developed to implement Floyd Warshall Algorithm:";

cout<<"\n================================================================\n";

cout<<"Read in Number of Vertices/Nodes in Graph: ";

cin>> n;

if (n <= 0 || n > MAXNODE)

{

cout<<"\nWrong Input! Please, Read in correct digit from 1 to 10:";

}

else

{

ReadNodes(Node, n); // Read in N nodes

ReadWeight(Weight, n, Node); //Read in weight/cost of each edges

DisplayWeight(Weight, n); // Display Matrix of Weight

ComputeShortestPath(Weight, n, Dist, Next);

DisplayShortestPath(Dist, n);

cout<<"\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ";

cout<<"Press 1 -> Yes \n Press 0 or any other digit -> No\n";

cin>>ch;

if(ch == 1) FindShortestPath(Dist, n , Node, Next);

}

}

void FloydWarshall::ReadNodes(char Nodes[], int N)

{

int i;

cout<<"Read in the names of all Nodes/Vertices in the graph\n";

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

{

cout<<"\nRead in the names of Node/Vertex "<<i+1<<":\t";

cin>>Nodes[i];

}

Nodes[i] = '\0';

}

void FloydWarshall::ReadWeight(int W[][MAXNODE], int N, char Nodes[])

{

int i, j;

cout<<"\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n";

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

{

cout<<"\n";

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

{

cout<<"Read in the weight/cost of edge between Node "<<Nodes[i]<<" and Node "<<Nodes[j]<<": ";

cin>>W[i][j];

}

}

}

void FloydWarshall::DisplayWeight(int W[][MAXNODE], int N)

{

int i, j;

cout<<"\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n";

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

{

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

{

cout<<W[i][j]<<"\t";

}

cout<<"\n";

}

}

void FloydWarshall::ComputeShortestPath(int edge[][MAXNODE], int M, int dist[][MAXNODE], int next[][MAXNODE])

{

int i, j, k, u, v;

/\* initialize dist \*/

for(u = 0; u < M; u++)

{

for(v = 0; v < M; v++)

{

if (edge[u][v] == 0) { dist[u][v] = INFINTY; next[u][v] = Null; }

else

{ dist[u][v] = edge[u][v]; next[u][v] = v; }

}

}

/\* Update dist with standard Floyd-Warshall implementation \*/

for(k = 0; k < M; k++)

{

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

{

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

{

if(dist[i][j] > dist[i][k] + dist[k][j])

{

dist[i][j] = dist[i][k] + dist[k][j];

next[i][j] = next[i][k];

}

}

}

}

}

void FloydWarshall::DisplayShortestPath(int dist[][MAXNODE], int M)

{

int i, j;

cout<<"\nThe Shortest Distance Table of all paths between 2 Vertices in the graph:\n";

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

{

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

{

if(dist[i][j] == INFINTY) cout<<"INF\t";

else

cout<<dist[i][j]<<'\t';

}

cout<<"\n";

}

}

void FloydWarshall::FindShortestPath(int dist[][MAXNODE], int M, char Nodes[], int next[][MAXNODE])

{

char source, target;

int u, j, v, k = 0, cost;

int \*path;

int found = FALSE;

path = (int\*)(malloc(M \* sizeof(int)));

cout<<"\nRead in the Source Node/Vertex: ";

cin>>source;

cout<<"\nRead in the Target Node/Vertex: ";

cin>>target;

/\* Verify source \*/

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

{

if(source == Nodes[j])

{

u = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ cout<<"\nWrong Input! Source Node not found; No path found: "; exit(0);}

found = FALSE;

/\* Verify target \*/

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

{

if(target == Nodes[j])

{

v = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ cout<<"\nWrong Input! Target Node not found; No path found: "; exit(0); }

cost = dist[u][v]; //get the shortest distance between source and target

//procedure to construct path

if(next[u][v] == Null)

{ cout<<"\n Failure, No path found! "; exit(0); }

path[k] = u;

while(u != v)

{

u = next[u][v];

path[++k] = u;

}

//display shortest path

cout<<"\n\nSuccessful, A Path Found!!! \nThe Shortest Path between "

<<source<<" and "<<target<<" is: "<<"\n "<<source<<"--->";

for(j = 1; j < k; j++) cout<<Node[path[j]]<<"--->";

cout<<Node[path[k]]<<"\n";

cout<<"The weight/cost of the shortest path between Node: "<<source<<" and Node: "

<<target<<" is: "<<cost<<"\n";

}

**RBC measure on Floyd – Warshall Algorithm implemented with C++**

1. **Input Output Complexity (IOC)**

No of input = Size (N) + Size (V) + Size (M) + Size (E) + Size (Dist) +

Size (Next) + Size (a) + Size (b) = 1 + 4 + 1 + 5 +16 + 16 + 1 + 1 = 45

No of output = Size (Dist) + Size (Path) + Size (c ) = 16 + 3 + 1 = 20

No of interfaces = 1 (console window)

No of files = 1 (Data storage)

IOC = No of input + No of output + No of Interface + No of files

= 45 + 20 + 1 + 1 = 67

1. **Functional Requirement (FR)**

No of functions = (ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), ComputeShortestPath ( ), DisplayShortestPath ( ) FindShortestPath ( )) = 7

**No of sub-process/sub- functions**

Function 1 - ShortestPath ( ) has 2 (Display, and Read Operations)

Function 2 - ReadNodes ( ) has 2 (Display, and Read operations)

Function 3 - ReadWeight ( ) has 2 (Display, and Read operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - ComputeShortestPath( ) has 1 (arithmetic addition)

Function 6 - DisplayShortestPath ( ) has 1 (Display operation)

Function 7 - FindShortestPath ( ) has 3 (Display, Read, and increment operations)

= 2 + 2 + 2 + 1 + 1 + 1 + 3 = 12

FR = Number of functions \* = 7 \* 12 = 84

**3. Non functional Requirement (NFR)**

NFR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 84 + 0 = 84

**5. Product Complexity (PC)**

PC = IOC \* RC = 67 \* 84 = 5628

**6. Personal Complexity Attributes (PCA)**

PCA = 0.90 (Suppose Programmer Capability = High)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((5628 \* 0.90) + 0 + 0 + 0 ) \* 1 = 5065.2

**3.4.3 Evaluation of Complexity of VB 6.0 Language Implementation of Floyd-Warshall Shortest path Algorithm**

' A VB 6.0 -PROGRAMMING LANGUAGE IMPLEMENTATION OF FLOYD WARSHALL SHORTEST PATH ALGORITHM

Const MAXNODE = 10

Const INFINTY = 9999

Const nulll = -1

'\*Procedure to compute shortest path weight for each edges

Private Sub ComputeShortestPath(ByRef edge() As Integer, ByRef dist() As Integer, ByRef Nextt() As Integer, ByVal n As Integer)

Dim i As Integer, j As Integer, k As Integer, u As Integer, v As Integer

'/\* initialize dist \*/

For u = 1 To n

For v = 1 To n

If (edge(u, v) = 0) Then

dist(u, v) = INFINTY

Nextt(u, v) = nulll

Else

dist(u, v) = edge(u, v): Nextt(u, v) = v

End If

Next v

Next u

'/\* Update dist with standard Floyd-Warshall implementation \*/

For k = 1 To n

For i = 1 To n

For j = 1 To n

If (dist(i, j) > dist(i, k) + dist(k, j)) Then

dist(i, j) = dist(i, k) + dist(k, j)

Nextt(i, j) = Nextt(i, k)

End If

Next j

Next i

Next k

End Sub

'\*Procedure for finding the shortest path \*

Private Sub FindShortestPath(ByRef dist() As Integer, ByRef Nodes() As String, ByRef Nextt() As Integer, ByVal M As Integer)

Dim source As String, j As Integer, u As Integer, v As Integer, target As String, cost As Integer

Const nulll As Integer = -1

Dim path() As Integer, k As Integer

Dim found As Boolean

found = False

'\* Dynamically Allocate Array \*

ReDim path(M)

'\*Read in source and target node

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the Source Node/Vertex: "

source = Left(InputBox("Read in the Source Node/Vertex: ", "Floyd Warshall"), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & source

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the Target Node/Vertex: "

target = Left(InputBox("Read in the target Node/Vertex: ", "Floyd Warshall"), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & target

'\* Verify source \*

For j = 1 To M

If (source = Nodes(j)) Then

u = j

found = True

'break

End If

Next j

If (found = False) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Source Node not found; No path found: "

Return

End If

found = False

'\* Verify target \*

For j = 1 To M

If (target = Nodes(j)) Then

v = j

found = True

'break

End If

Next j

If (found = False) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Target Node not found; No path found: "

Return

End If

cost = dist(u, v) 'get the shortest distance between the source and target

'procedure to construct path

If (Nextt(u, v) = nulll) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Failure, No path found! " & vbCrLf

Return

End If

'//getting the shortest path

k = 1

path(k) = u

While (u <> v) '//construct the shortest path with a stack S

u = Nextt(u, v)

k = k + 1: path(k) = u

Wend

' //display shortest path

TxtOutput.Text = TxtOutput.Text & vbCrLf & vbCrLf & "Successful, A Path Found!!! \nThe Shortest Path between " + source + " and " + target + " is: " & vbCrLf + source + "--->"

For j = 2 To k - 1

TxtOutput.Text = TxtOutput.Text & Nodes(path(j)) & "--->"

Next j

TxtOutput.Text = TxtOutput.Text & Nodes(path(k))

TxtOutput.Text = TxtOutput.Text & vbCrLf & "The weight/cost of the shortest path between Node: " & source & " and Node: " & target & " is " & cost

End Sub

'\*Procedure for reading Nodes \*

Private Sub readNodes(ByRef Nodes() As String, ByVal M As Integer)

Dim j As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the names of all Nodes/Vertices in the graph"

MsgBox "Read in the names of all Nodes/Vertices in the graph", vbOKOnly, "Floyd Warshall"

ReDim Nodes(M)

For j = 1 To M

Nodes(j) = InputBox("Read in the names of Node/Vertex " & j & " : ", "Floyd Warshall")

Nodes(j) = Left$(Nodes(j), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the names of Node/Vertex " & j & " : " & Nodes(j)

Next j

End Sub

'\*Procedure for reading weight \*

Private Sub readWeight(ByRef W() As Integer, ByRef Nodes() As String, ByVal n As Integer)

Dim j As Integer, i As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the weights/costs of all edges between 2 Vertices in the graph"

MsgBox "Read in the weights/costs of all edges between 2 Vertices in the graph:", vbOKOnly, "Floyd Warshall"

For i = 1 To n

For j = 1 To n

W(i, j) = Val(InputBox("Read in the weight/cost of edge between Node " & Nodes(i) & " and Node " & Nodes(j) & ": ", "Floyd Warshall"))

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the weight/cost of edge between Node " & Nodes(i) & " and Node " & Nodes(j) & ": " & W(i, j)

' If (W(i, j) = 0) Then W(i, j) = INFINTY

Next j

Next i

End Sub

'\*Procedure to generate Itemset \*

'\* Procedure to display the weight/cost table\*

Private Sub DisplayWeight(ByRef W() As Integer, ByVal n As Integer)

Dim i As Integer, j As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "The weights/costs Table of all edges between 2 Vertices in the graph: "

For i = 1 To n

TxtOutput.Text = TxtOutput.Text & vbCrLf

For j = 1 To n

TxtOutput.Text = TxtOutput.Text & W(i, j) & " "

Next j

Next i

End Sub

'\* Procedure to display the shortest path weight/cost table\*

Private Sub DisplayShortestPath(ByRef dist() As Integer, ByVal n As Integer)

Dim i As Integer, j As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "The Shortest path costs Table of all edges between 2 Vertices in the graph: "

For i = 1 To n

TxtOutput.Text = TxtOutput.Text & vbCrLf

For j = 1 To n

If (dist(i, j) = INFINTY) Then

TxtOutput.Text = TxtOutput.Text & "INF "

Else

TxtOutput.Text = TxtOutput.Text & dist(i, j) & " "

End If

Next j

Next i

End Sub

Private Sub CmdCancel\_Click()

End

End Sub

Private Sub CmdReset\_Click()

TxtOutput.Text = ""

CmdStart.Enabled = True

End Sub

Private Sub CmdStart\_Click()

'\*Begin the main procedure for implementing Floyd Warshall Algorithm \*

Dim Node() As String, Weight(MAXNODE, MAXNODE) As Integer, dist(MAXNODE, MAXNODE) As Integer, Nextt(MAXNODE, MAXNODE) As Integer

Dim noOfNodes As Integer, Ch As Integer, source As String \* 1, target As String \* 1

TxtOutput.Text = TxtOutput.Text & "Welcome to VB 6.0 Program developed to implement Floyd Warshall Algorithm:"

TxtOutput.Text = TxtOutput.Text & vbCrLf & "...................................................................."

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in Number of Vertices/Nodes in Graph: "

noOfNodes = Val(InputBox("Read in Number of Vertices/Nodes in Graph: ", "Floyd Warshall"))

TxtOutput.Text = TxtOutput.Text + Str(noOfNodes)

If (noOfNodes <= 0 Or noOfNodes > MAXNODE) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Please, Read in correct digit from 1 to 10:"

Else

Call readNodes(Node, noOfNodes)

Call readWeight(Weight, Node, noOfNodes)

Call DisplayWeight(Weight, noOfNodes)

Call ComputeShortestPath(Weight, dist, Nextt, noOfNodes)

Call DisplayShortestPath(dist, noOfNodes)

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Do you wish to find Shortest Path between two Vertices/Nodes in the Graph: "

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Press 1 -> Yes"

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Press 0 or any other digit -> No"

Ch = Val(InputBox("Press 1 -> Yes, Press 0 or any other digit -> No", "Floyd Warshall"))

TxtOutput.Text = TxtOutput.Text & vbCrLf & Str(Ch)

If (Ch = 1) Then

Call FindShortestPath(dist, Node, Nextt, noOfNodes)

End If

End If

TxtOutput.Text = TxtOutput.Text & vbCrLf & "\*\*\*\*\*\* Press any Key to Continue \*\*\*\*\*\*"

CmdStart.Enabled = False

End Sub

**RBC measure on Floyd – Warshall Algorithm implemented with VB 6.0**

1. **Input Output Complexity (IOC)**

No of input = Size (N) + Size (V) + Size (M) + Size (E) + Size (Dist) + Size (Next)+ Size (a) + Size (b) = 1 + 4 + 1 + 5 +16 + 16 + 1 + 1 = 45

No of output = Size (Dist) + Size (Path) + Size (c ) = 16 + 3 + 1 = 20

No of interface = 1 (console window)

No of fites = 1 (Data storage)

IOC = No of input + No of output + No of Interface + No of files

= 45 + 20 + 1 + 1 = 67

1. **Functional Requirement (FR)**

No of functions = (ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), ComputeShortestPath ( ), DisplayShortestPath ( ) FindShortestPath ( )) = 6

**No of sub-process/sub- functions**

Function 1 - ReadNodes ( ) has 2 (Display, and Read operations)

Function 2 - ReadWeight ( ) has 2 (Display, and Read operations)

Function 3 - DisplayWeight ( ) has 1 (Display operations)

Function 4 - ComputeShortestPath( ) has 1 (arithmetic addition)

Function 5 - DisplayShortestPath ( ) has 1 (Display operation)

Function 6 - FindShortestPath ( ) has 3 (Display, Read, and increment operations)

= 2 + 2 + 1 + 1 + 1 + 3 = 10

FR = Number of functions \* = 6 \* 10 = 60

**3. Non functional Requirement (NFR)**

NFR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 60 + 0 = 60

**5. Product Complexity (PC)**

PC = IOC \* RC = 67 \* 60 = 4020

**6. Personal Complexity Attributes (PCA)**

PCA = 0.90 (Suppose Programmer Capability = High)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1, Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((4020 \* 0.90) + 0 + 0 + 0 ) \* 1 = 3618

**3.4.4 Evaluation of Complexity of JAVA Language Implementation of Floyd-Warshall Shortest path Algorithm**

/\* A JAVA Program to compute the Shortest Path using Floyd Warshall Algorithm

\* Class Name: FloydWarshall.java

\* Project Developed by

\*/

import java.util.Scanner;

public class FloydWarshall {

private final int MAXNODE = 10;

private final int TRUE = 1;

private final int FALSE = 0;

private final int INFINTY = 9999;

private final int NULL = -1;

private int[][] Weight;

private int[][] Dist ;

private int[][] Next ;

public int noOfNodes;

private char[] Node;

Scanner scanInput = new Scanner(System.in);

public FloydWarshall(int N)

{

setNoOfNodes(N);

Weight = new int[N][N];

Dist = new int[N][N];

Next = new int[N][N];

Node = new char[N];

}

private int getNoOfNodes(){

return noOfNodes;

}

private void setNoOfNodes(int Nnodes){

noOfNodes = Nnodes;

}

public void ShortestPath()

{

int ch;

if (noOfNodes <= 0 || noOfNodes > MAXNODE)

{

System.out.println("Wrong Input! Please, Read in correct digit from 1 to 10:");

}

else

{

ReadNodes(Node); // Read in N nodes

ReadWeight(Weight, Node); //Read in weight/cost of each edges

DisplayWeight(Weight); // Display Matrix of Weight

ComputeShortestPath(Weight, Dist, Next);

DisplayShortestPath(Dist);

System.out.println("\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ");

System.out.println("Press 1 -> Yes \n Press 0 or any other digit -> No\n");

ch = scanInput.nextInt();

if(ch == 1) FindShortestPath(Dist, Node, Next);

}

}

private void ReadNodes(char[] Nodes)

{

int i;

System.out.println("Read in the names of all Nodes/Vertices in the graph");

for(i = 0; i < Nodes.length; i++)

{

System.out.print("\nRead in the names of Node/Vertex " + (i+1) + ":\t" );

String Str = scanInput.next();

Nodes[i] = Str.charAt(0);

}

}

private void ReadWeight(int[][] W, char[] Nodes)

{

int i, j;

System.out.print("\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n");

for(i = 0; i < W.length; i++)

{

System.out.println();

for (j = 0; j < W[i].length; j++)

{

System.out.print("Read in the weight/cost of edge between Node "+ Nodes[i]

+ " and Node " + Nodes[j] + ": ");

W[i][j] = scanInput.nextInt();

}

}

}

private void DisplayWeight(int[][] W)

{

int i, j;

System.out.print("\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n");

for (i = 0; i < W.length; i++)

{

for (j = 0; j < W[i].length; j++)

{

System.out.printf("%6d", W[i][j]);

}

System.out.print("\n");

}

}

private void ComputeShortestPath(int edge[][], int dist[][], int next[][])

{

int i, j, k, u, v;

/\* initialize dist \*/

for(u = 0; u < edge.length; u++)

{

for(v = 0; v < edge[u].length; v++)

{

if (edge[u][v] == 0) { dist[u][v] = INFINTY; next[u][v] = NULL; }

else

{ dist[u][v] = edge[u][v]; next[u][v] = v; }

}

}

/\* Update dist with standard Floyd-Warshall implementation \*/

for(k = 0; k < dist.length; k++)

{

for(i = 0; i < dist.length; i++)

{

for(j = 0; j < dist.length; j++)

{

if(dist[i][j] > dist[i][k] + dist[k][j])

{

dist[i][j] = dist[i][k] + dist[k][j];

next[i][j] = next[i][k];

}

}

}

}

}

private void DisplayShortestPath(int[][] dist)

{

int i, j;

System.out.print("\nThe Shortest Path Table of all path between 2 Vertices in the graph:\n");

for (i = 0; i < dist.length; i++)

{

for (j = 0; j < dist[i].length; j++)

{

if(dist[i][j] == INFINTY) System.out.print(" INF");

else

System.out.printf("%6d", dist[i][j]);

}

System.out.print("\n");

}

}

private void FindShortestPath(int[][] dist, char Nodes[], int[][] next)

{

char source, target;

int u = 0, j, v = 0, k = 0, cost;

int[] path = new int[Nodes.length];

int found;

found = FALSE;

System.out.print("\nRead in the Source Node/Vertex: ");

String str = scanInput.next();

source = str.charAt(0);

System.out.print("\nRead in the Target Node/Vertex: ");

str = scanInput.next();

target = str.charAt(0);

/\* Verify source \*/

for (j = 0; j < Nodes.length; j++)

{

if(source == Nodes[j])

{

u = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ System.out.print("\nWrong Input! Source Node not found; No path found: "); return;}

found = FALSE;

/\* Verify target \*/

for(j = 0; j < Nodes.length; j++)

{

if(target == Nodes[j])

{

v = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ System.out.print("\nWrong Input! Target Node not found; No path found: "); return; }

cost = dist[u][v]; //get the shortest distance between source and target

//procedure to construct path

if(next[u][v] == NULL)

{ System.out.print("\n Failure, No path found! "); return; }

path[k] = u;

while(u != v)

{

u = next[u][v];

path[++k] = u;

}

//display shortest path

System.out.print("\n\nSuccessful, A Path Found!!! \nThe Shortest Path between " +

source + " and " + target + " is:\n " + source + "--->");

for(j = 1; j < k; j++) System.out.print(Node[path[j]]+ "--->");

System.out.print(Node[path[k]]+ "\n");

System.out.print("\nThe weight/cost of the shortest path between Node: " +

source + " and Node: " + target + " is " + cost + "\n");

}

}

/\* A Java Program to compute the Shortest Path using Floyd Warshall Algorithm

\* This is a main Class name: FloydWarshallShortestPath.java

\* Project Developed by

\*/

import java.util.Scanner;

public class FloydWarshallShortestPath {

public static void main(String[] args)

{

Scanner scanInput = new Scanner(System.in);

int n;

System.out.println("Welcome to Java Program developed to implement Floyd Warshall Algorithm:");

System.out.println("=================================================================\n");

System.out.print("Read in Number of Vertices/Nodes in Graph: ");

n = scanInput.nextInt();

FloydWarshall floydWarshall = new FloydWarshall(n);

floydWarshall.ShortestPath();

}

}

**RBC measure on Floyd – Warshall Algorithm implemented with JAVA**

1. **Input Output Complexity (IOC)**

No of input = Size (N) + Size (V) + Size (M) + Size (E) + Size (Dist) + Size (Next) + Size (a) + Size (b) = 1 + 4 + 1 + 5 +16 + 16 + 1 + 1 = 45

No of output = Size (Dist) + Size (Path) + Size (c ) = 16 + 3 + 1 = 20

No of interface = 1 (console window)

No of fites = 1 (Data storage)

IOC = No of input + No of output + No of Interface + No of files

= 45 + 20 + 1 + 1 = 67

1. **Functional Requirement (FR)**

No of functions = (FloydWarshellShortestPath ( ), SetNoOfNodes ( ), GetNoOfNodes ( ), ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), ComputeShortestPath ( ), DisplayShortestPath ( ) FindShortestPath ( ) = 10

**No of sub-process/sub- functions**

Function 1 - ShortestPath ( ) has 3 (Display, Read, Type-Casting operations)

Function 2 - ReadNodes ( ) has 3 (Display, Read, Type-Casting operations)

Function 3 - ReadWeight ( ) has 3 (Display, Read, Type-casting operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - ComputeShortestPath( ) has 1 (arithmetic addition)

Function 6 - DisplayShortestPath ( ) 1 (Display operation)

Function 7 - FindShortestPath ( ) has 4 (Display, Read, Type-Casting & increment operations)

= 3 + 3 + 3 + 1 + 1 + 1 + 4 = 16

FR = Number of functions \* = 10 \* 16 = 160

**3. Non functional Requirement (NFR)**

NFR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 160 + 0 = 160

**5. Product Complexity (PC)**

PC = IOC \* RC = 67 \* 160 = 10720

**6. Personal Complexity Attributes (PCA)**

PCA = 1.00 (Suppose Programmer Capability = Normal)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((10720 \* 1.00) + 0 + 0 + 0 ) \* 1 = 10720

**3.5 Empirical analysis and comparism of Implemented Bellman Ford algorithm using Requirement Based Complexity (RBC) Measures.**

Complexity of Bellman Ford algorithm is the amount of resources necessary to execute it. The following are the RBC values of Bellman Ford algorithm when implemented in C#, C++, Java and Visual BASIC programming languages, and tested with the graph shown in figure 3.3:



Fig. 3.3.A diagrammatic representation of a typicalGraph G (V, E)

**3.5.1 Evaluation of Complexity of C# Language Implementation of Bellman Ford Shortest path Algorithm**

/\* A C# Program to compute the Shortest Path using Bellman Ford Algorithm

\* This is a Class name: BellmanFordShortestPath.cs

\* Project Developed by

\*/

using System;

namespace BellmanFordShortestPathProject

{

class BellmanFordShortestPath

{

private const int MAXNODE = 10;

private const int TRUE = 1;

private const int FALSE = 0;

private const int INFINTY = 9999;

private int[,] Edge;

public int noOfNodes;

private char[] Vertice;

public BellmanFordShortestPath(int N)

{

setNoOfNodes(N);

Edge = new int[N,N];

Vertice = new char[N];

}

private int getNoOfNodes(){

return noOfNodes;

}

private void setNoOfNodes(int Nnodes){

noOfNodes = Nnodes;

}

public void ShortestPath()

{

int ch;

if (noOfNodes <= 0 || noOfNodes > MAXNODE)

{

Console.WriteLine("Wrong Input! Please, Read in correct digit from 1 to 10:");

}

else

{

ReadNodes(Vertice); // Read in N nodes

ReadWeight(Edge, Vertice); //Read in weight/cost of each edges

DisplayWeight(Edge); // Display Matrix of Weight

Console.WriteLine("\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ");

Console.WriteLine("Press 1 -> Yes \n Press 0 or any other digit -> No\n");

ch = Convert.ToInt32(Console.ReadLine());

if(ch == 1)

{

Console.Write("\nRead in the Source Node/Vertex: ");

char source = Convert.ToChar(Console.ReadLine());

Console.Write("\nRead in the Target Node/Vertex: ");

char target = Convert.ToChar(Console.ReadLine());

FindShortestPath(Vertice, Edge, source, target);

}

}

}

private void ReadNodes(char[] Nodes)

{

int i;

Console.WriteLine("Read in the names of all Nodes/Vertices in the graph");

for(i = 0; i < Nodes.Length; i++)

{

Console.Write("\nRead in the names of Node/Vertex " + (i+1) + ":\t" );

Nodes[i] = Convert.ToChar(Console.ReadLine());

}

}

private void ReadWeight(int[,] W, char[] Nodes)

{

int i, j;

Console.Write("\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n");

for (i = 0; i < W.GetLength(0); i++)

{

Console.WriteLine();

for (j = 0; j < W.GetLength(1); j++)

{

Console.Write("Read in the weight/cost of edge between Node " + Nodes[i]

+ " and Node " + Nodes[j] + ": ");

W[i, j] = Convert.ToInt32(Console.ReadLine());

if (W[i, j] == 0) W[i, j] = INFINTY;

}

}

}

private void DisplayWeight(int[,] W)

{

int i, j;

Console.Write("\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n");

for (i = 0; i < W.GetLength(0); i++)

{

for (j = 0; j < W.GetLength(1); j++)

{

if (W[i, j] == INFINTY) Console.Write(" INF");

else

Console.Write("{0,6}", W[i, j]);

}

Console.WriteLine();

}

}

private void FindShortestPath(char[] Vertices, int[,] Edges, char source, char target)

{

int souce = 0, j, u, v, taget = 0, i, w, Null = -1;

int[] distance, predecessor, S;

int found = FALSE;

/\* Dynamically Allocate Array \*/

distance = new int[Vertices.Length];

predecessor = new int[Vertices.Length];

S = new int[Vertices.Length];

/\* Verify source \*/

for (j = 0; j < Vertices.Length; j++)

{

if(source == Vertices[j])

{

souce = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ Console.Write("\nWrong Input! Source Node not found; No path found: "); return;}

found = FALSE;

/\* Verify target \*/

for(j = 0; j < Vertices.Length; j++)

{

if(target == Vertices[j])

{

taget = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ Console.Write("\nWrong Input! Target Node not found; No path found: "); return; }

//Implementing Bellman Ford Algorithm

//Step 1: initialize graph

for(v = 0; v < Vertices.Length; v++)

{

if(v == souce) distance[v] = 0; //distance from source to source

else

distance[v] = INFINTY; //unknown distance function from source to v

predecessor[v] = Null; //predecessor node in optimal path from source

}

//Step 2: relax edges repeatedly

for(i = 0; i < Vertices.Length-1; i++)

{

for(u = 0; u < Vertices.Length; u++)

{

for(v = 0; v < Vertices.Length; v++)

{

w = Edges[u,v];

if((distance[u] + w) < distance[v])

{

distance[v] = distance[u] + w;

predecessor[v] = u;

}

}

}

}

//return distance[], predecessor[]

Console.Write("\nDistance table: \n") ;

for(v = 0; v < Vertices.Length; v++)

{

if(distance[v] == INFINTY) Console.Write("INF\t");

else

Console.Write(distance[v]+" \t");

}

Console.Write("\nPredecessor table: \n");

for(v = 0; v < Vertices.Length; v++) Console.Write(predecessor[v]+1 + " \t");

//getting the shortest path

j = 0;

u = taget;

while(predecessor[u] != Null) //construct the shortest path with a stack S

{

S[j++] = u; //Push the vertex onto the stack

u = predecessor[u]; //Traverse from target to source

}

//display shortest path

Console.Write("\n\nSuccessful, A Path Found!!! \nThe Shortest Path between " +

source + " and " + target + " is:\n " + source + "--->");

for(v = j-1; v > 0; v--) Console.Write(Vertices[S[v]]+ "--->");

Console.Write(Vertices[S[0]]+ "\n");

Console.Write("\nThe weight/cost of the shortest path between Node: " +

source + " and Node: " + target + " is " + distance[taget] + "\n");

}

}

}

/\* A C# Program to compute the Shortest Path using Bellman Ford Algorithm

\* This is a main Class name: BellmanFordShortestPathMain.cs

\* Project Developed by

\*/

using System;

namespace BellmanFordShortestPathProject

{

class BellmanFordShortestPathMain

{

static void Main(string[] args)

{

int n;

Console.WriteLine("Welcome to C# Program developed to implement Bellman Ford Algorithm:");

Console.WriteLine("=================================================================\n");

Console.Write("Read in Number of Vertices/Nodes in Graph: ");

n = Convert.ToInt32(Console.ReadLine());

BellmanFordShortestPath bellmanFord = new BellmanFordShortestPath(n);

bellmanFord.ShortestPath();

}

}

}

**RBC measure on Bellman Ford Algorithm implemented with C#**

**Input**

Extracting from the graph in fig. 3.3 above:

N = Number of vertices V = 6, count (N) = 1

V = { S, A, B, C, D, t } , count = 6

M = Number of Edges = 9, count (M) = 1

E = Edges map { 5, -2, 2, 1, 2, 3, 7, 3, 10}, count (E) = 9

Distance = vertex distance map of size |V| which must be initialized to ∞ for all vertices u in the graph, except for source vertex S, distance(s) = 0

Count (distance) = N = 6

Predecessor = A predecessor map (option 4) of size N, which must be initialized to Null count (distance) = N = 6

s = Source vertex, count(s) = 1

t = Target vertex, count(t) = 1

No of input = count (N) + count (V) + count (M) + count (E) + count (Distance) + count (predecessor) + count (S) + count (t)

= 1 + 6 + 1 + 9 + 6 + 6 + 1 + 1 = 31

**Output**

* Distance = |V| array of the computed shortest distance from the Source to every vertex in the graph

Count (Distance) = 6

* The shortest path constructed from a given source s to the target t

Path = { S, E, A, B, t }, Count (path) = 5

* The cost/weight of the shortest path found, computed as;

costs = -2 + 2 + 1 + 3 = 4, count (cost ) = 1

* No of output = count (Distance) + count (Path) + count (cost)

= 6 + 5 + 1 = 12

* No of interface = 1 (console window)
* No of files = 1 (Data storage)

1. **Input Output Complexity (IOC)**

IOC = No of input + No of output + No of Interface + No of files

= 31 + 12 + 1 + 1 = 45

1. **Functional Requirement (FR)**

No of functions = (BellManFordShortestPath ( ), SetNoOfNodes ( ), GetNoOfNodes ( ), ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), FindShortestPath ( )) = 8

No of sub-process/sub-functions ⇒ Every function Fi has zero, one or more

sub- processes such as Arithmetic, Computation, Display, etc.

Function 1 - ShortestPath ( ) has 3 (Display operations, Read operations, Type-Casting operations)

Function 2 - ReadNodes ( ) has 3 (Display operations, Read operations, Type-Casting operations)

Function 3 - ReadWeight ( ) has 3 (Display, Read, Type-casting operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - FindShortestPath ( ) 2 (Display, arithmetic addition)

= 3 + 3 + 3 + 1 + 2 = 12

FR = Number of functions \* = 8 \* 12 = 96

**3. Non functional Requirement (NFR)**

Number of Non-FR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 96 + 0 = 96

**5. Product Complexity (PC)**

PC = IOC \* RC = 45 \* 96 = 4320

**6**. **Personal Complexity Attributes (PCA)**

PCA = 1.17 (Suppose Programmer Capability = Low)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((4320 \* 1.17) + 0 + 0 + 0 ) \* 1 = 5054.4

**3.5.2 Evaluation of Complexity of C++ Language Implementation of Bellman Ford Shortest path Algorithm**

/\* A C++ Program to compute the Shortest Path using Bellman Ford Algorithm

\* Project Developed by

\*/

#include <cstdlib>

#include <iostream>

#include <conio.h>

#include <stdlib.h>

#include <ctype.h>

#include <math.h>

#define MAXNODE 10

#define TRUE 1

#define FALSE 0

#define INFINTY 9999

using std::cout;

using std::cin;

class BellmanFord {

private:

int Edges[MAXNODE][MAXNODE], n, ch;

char Vertices[MAXNODE];

public:

void ReadNodes(char Nodes[], int N);

void ReadWeight(int W[][MAXNODE], int N, char Nodes[]);

void DisplayWeight(int W[][MAXNODE], int N);

void BellmanFordShortestPath(char Vertices[], int Edges[][MAXNODE], int M, char source, char target);

void ShortestPath();

};

int main()

{

BellmanFord bellmanFord;

bellmanFord.ShortestPath();

system("PAUSE");

return 0;

}

void BellmanFord::ShortestPath()

{

char source, target;

cout<<"Welcome to C++ Program developed to implement Bellman Ford Algorithm:";

cout<<"\n================================================================\n";

cout<<"Read in Number of Vertices/Nodes in Graph: ";

cin>> n;

if (n <= 0 || n > MAXNODE)

{

cout<<"\nWrong Input! Please, Read in correct digit from 1 to 10:";

}

else

{

ReadNodes(Vertices, n); // Read in N nodes

ReadWeight(Edges, n, Vertices); //Read in weight/cost of each edges

DisplayWeight(Edges, n); // Display Matrix of Weight

cout<<"\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ";

cout<<"Press 1 -> Yes \n Press 0 or any other digit -> No\n";

cin>>ch;

if(ch == 1)

{

cout<<"\nRead in the Source Node/Vertex: "; cin>>source;

cout<<"\nRead in the Target Node/Vertex: "; cin>>target;

BellmanFordShortestPath(Vertices, Edges, n, source, target);

} //end if

} //end else

}

void BellmanFord::ReadNodes(char Nodes[], int N)

{

int i;

cout<<"Read in the names of all Nodes/Vertices in the graph\n";

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

{

cout<<"\nRead in the names of Node/Vertex "<<i+1<<":\t";

cin>>Nodes[i];

}

Nodes[i] = '\0';

}

void BellmanFord::ReadWeight(int W[][MAXNODE], int N, char Nodes[])

{

int i, j;

cout<<"\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n";

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

{

cout<<"\n";

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

{

cout<<"Read in the weight/cost of edge between Node "<<Nodes[i]<<" and Node "<<Nodes[j]<<": ";

cin>>W[i][j];

if (W[i][j] == 0) W[i][j] = INFINTY;

}

}

}

void BellmanFord::DisplayWeight(int W[][MAXNODE], int N)

{

int i, j;

cout<<"\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n";

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

{

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

{

if(W[i][j] == INFINTY) cout<<"INF\t";

else

cout<<W[i][j]<<"\t";

}

cout<<"\n";

}

}

void BellmanFord::BellmanFordShortestPath(char Vertices[], int Edges[][MAXNODE], int M, char source, char target)

{

int souce, j, alt, u, v, taget, i, w;

int \*distance, \*predecessor, \*S;

int found = FALSE, null = -1;

/\* Dynamically Allocate Array \*/

distance = (int \*) (malloc(M \* sizeof(int)));

predecessor = (int \*) (malloc(M \* sizeof(int)));

S = (int \*) (malloc(M \* sizeof(int)));

/\* Verify source \*/

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

{

if(source == Vertices[j])

{

souce = j; //get the index position of source from the Node

found = TRUE;

break;

}

}

if (found == FALSE)

{ cout<<"\nWrong Input! Source Node not found; No path found: "; exit(0);}

found = FALSE;

/\* Verify target \*/

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

{

if(target == Vertices[j])

{

taget = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ cout<<"\nWrong Input! Target Node not found; No path found: "; exit(0); }

//Implementing Bellman Ford Algorithm

//Step 1: initialize graph

for(v = 0; v < M; v++)

{

if(v == souce) distance[v] = 0; //distance from source to source

else

distance[v] = INFINTY; //unknown distance function from source to v

predecessor[v] = null; //predecessor node in optimal path from source

}

//Step 2: relax edges repeatedly

for(i = 0; i < M-1; i++)

{

for(u = 0; u < M; u++)

{

for(v = 0; v < M; v++)

{

w = Edges[u][v];

if((distance[u] + w) < distance[v])

{

distance[v] = distance[u] + w;

predecessor[v] = u;

}

}

}

}

//return distance[], predecessor[]

cout << "\nDistance table: \n" ;

for(v = 0; v < M; v++)

{

if(distance[v] == INFINTY) cout<<"INF\t";

else

cout<<distance[v]<<" \t";

}

cout<<"\nPredecessor table: \n";

for(v = 0; v < M; v++) cout<<predecessor[v]+1<<" \t";

//getting the shortest path

j = 0;

u = taget;

while(predecessor[u] != null) //construct the shortest path with a stack S

{

S[j++] = u; //Push the vertex onto the stack

u = predecessor[u]; //Traverse from target to source

}

//display shortest path

cout<<"\n\nSuccessful, A Path Found!!! \nThe Shortest Path between "

<<source<<" and "<<target<<" is: "<<"\n "<<source<<"--->";

for(v = j-1; v > 0; v--) cout<<Vertices[S[v]]<<"--->";

cout<<Vertices[S[0]]<<"\n";

cout<<"The weight/cost of the shortest path between Node: "<<source<<" and Node: "

<<target<<" is: "<<distance[taget]<<"\n";

}

**RBC measure on Bellman Ford Algorithm implemented with C++**

1. **Input Output Complexity (IOC)**

No of input = count (N) + count (V) + count (M) + count (E) + count (Distance) + count (predecessor) + count (S) + count (t)

= 1 + 6 + 1 + 9 + 6 + 6 + 1 + 1 = 31

* No of output = count (Distance) + count (Path) + count (cost)

= 6 + 5 + 1 = 12

* No of interface = 1 (console window)
* No of files = 1 (Data storage)

IOC = No of input + No of output + No of Interface + No of files

= 31 + 12 + 1 + 1 = 45

1. **Functional Requirement (FR)**

No of functions = (BellManFordShortestPath ( ), ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( )) = 5

No of sub-process/sub-functions ⇒ Every function Fi has zero, one or more

sub-processes such as Arithmetic, Computation, Display, etc.

Function 1 - ShortestPath ( ) has 2 (Display operations, Read operations)

Function 2 - ReadNodes ( ) has 2 (Display operations, Read operations)

Function 3 - ReadWeight ( ) has 2 (Display, Read operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - BellmanFordShortestPath ( ) has 2 (Display, arithmetic addition)

= 2 + 2 + 2 + 1 + 2 = 9

FR = Number of functions \* = 5 \* 9 = 45

**3. Non functional Requirement (NFR)**

Number of Non-FR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 45 + 0 = 45

**5. Product Complexity (PC)**

PC = IOC \* RC = 45 \* 45 = 2025

**6**. **Personal Complexity Attributes (PCA)**

PCA = 0.90 (Suppose Programmer Capability = High)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((2025 \* 0.90) + 0 + 0 + 0 ) \* 1 = 1822.5

**3.5.3 Evaluation of Complexity of VB 6.0 Language Implementation of Bellman Ford Shortest path Algorithm**

' A VB 6.0 -PROGRAMMING LANGUAGE IMPLEMENTATION OF BELLMAN FORD SHORTEST PATH ALGORITHM

Const MAXNODE = 10

Const INFINTY = 9999

'\*Procedure for finding the shortest path \*

Private Sub FindShortestPath(ByRef Vertice() As String, ByRef Edge() As Integer, ByVal source As String, ByVal target As String, ByVal M As Integer)

Dim souce As Integer, j As Integer, u As Integer, v As Integer, taget As Integer, w As Integer

Const Nulll As Integer = -1

Dim Distance() As Integer, predecessor() As Integer, S() As Integer

Dim found As Boolean

found = False

'\* Dynamically Allocate Array \*

ReDim Distance(M)

ReDim predecessor(M)

ReDim S(M)

'\* Verify source \*

For j = 1 To M

If (source = Vertice(j)) Then

souce = j

found = True

'break

End If

Next j

If (found = False) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Source Node not found; No path found: "

Return

End If

found = False

'\* Verify target \*

For j = 1 To M

If (target = Vertice(j)) Then

taget = j

found = True

'break

End If

Next j

If (found = False) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Target Node not found; No path found: "

Return

End If

For v = 1 To M

If (v = souce) Then

Distance(v) = 0 'distance from source to source

Else

Distance(v) = INFINTY 'unknown distance function from source to v

End If

predecessor(v) = Nulll 'previous node in optimal path initialization

Next v

'//step 2: relax edges repeatedly

For i = 1 To M - 1

For u = 1 To M

For v = 1 To M

w = Edge(u, v)

If ((Distance(u) + w) < (Distance(v))) Then

Distance(v) = Distance(u) + w

predecessor(v) = u

End If

Next v

Next u

Next i

'//return distance[], predecessor[]

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Distance table: " & vbCrLf

For v = 1 To M

If (Distance(v) = INFINTY) Then

TxtOutput.Text = TxtOutput.Text & "INF "

Else

TxtOutput.Text = TxtOutput.Text & Distance(v) & " "

End If

Next v

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Predecessor table: " & vbCrLf

For v = 1 To M

TxtOutput.Text = TxtOutput.Text & predecessor(v) & " "

Next v

'//getting the shortest path

j = 1

u = taget

While (predecessor(u) <> Nulll) '//construct the shortest path with a stack S

S(j) = u: j = j + 1 ' //Push the vertex onto the stack

u = predecessor(u) ' //Traverse from target to source

Wend

' //display shortest path

TxtOutput.Text = TxtOutput.Text & vbCrLf & vbCrLf & "Successful, A Path Found!!! \nThe Shortest Path between " + source + " and " + target + " is: " & vbCrLf + source + "--->"

For v = j - 1 To 2 Step -1

TxtOutput.Text = TxtOutput.Text & Vertice(S(v)) & "--->"

Next v

TxtOutput.Text = TxtOutput.Text & Vertice(S(1))

TxtOutput.Text = TxtOutput.Text & vbCrLf & "The weight/cost of the shortest path between Node: " & source & " and Node: " & target & " is " & Distance(taget)

End Sub

'\*Procedure for reading Nodes \*

Private Sub readNodes(ByRef Nodes() As String, ByVal M As Integer)

Dim j As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the names of all Nodes/Vertices in the graph"

MsgBox "Read in the names of all Nodes/Vertices in the graph", vbOKOnly, "Bellman Ford"

ReDim Nodes(M)

For j = 1 To M

Nodes(j) = InputBox("Read in the names of Node/Vertex " & j & " : ", "Bellman Ford")

Nodes(j) = Left$(Nodes(j), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the names of Node/Vertex " & j & " : " & Nodes(j)

Next j

End Sub

'\*Procedure for reading weight \*

Private Sub readWeight(ByRef w() As Integer, ByRef Nodes() As String, ByVal n As Integer)

Dim j As Integer, i As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the weights/costs of all edges between 2 Vertices in the graph"

MsgBox "Read in the weights/costs of all edges between 2 Vertices in the graph:", vbOKOnly, "DIJKSTRA"

For i = 1 To n

For j = 1 To n

w(i, j) = Val(InputBox("Read in the weight/cost of edge between Node " & Nodes(i) & " and Node " & Nodes(j) & ": ", "Dijkstra"))

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the weight/cost of edge between Node " & Nodes(i) & " and Node " & Nodes(j) & ": " & w(i, j)

If (w(i, j) = 0) Then w(i, j) = INFINTY

Next j

Next i

End Sub

Private Sub DisplayWeight(ByRef w() As Integer, ByVal n As Integer)

Dim i As Integer, j As Integer

TxtOutput.Text = TxtOutput.Text & vbCrLf & "The weights/costs Table of all edges between 2 Vertices in the graph: "

For i = 1 To n

TxtOutput.Text = TxtOutput.Text & vbCrLf

For j = 1 To n

If (w(i, j) = INFINTY) Then

TxtOutput.Text = TxtOutput.Text & "INF "

Else

TxtOutput.Text = TxtOutput.Text & w(i, j) & " "

End If

Next j

Next i

End Sub

Private Sub CmdCancel\_Click()

End

End Sub

Private Sub CmdReset\_Click()

TxtOutput.Text = ""

CmdStart.Enabled = True

End Sub

Private Sub CmdStart\_Click()

'\*Begin the main procedure for implementing Bellman Fords Algorithm \*

Dim Vertices() As String, Edges(MAXNODE, MAXNODE) As Integer

Dim noOfNodes As Integer, Ch As Integer, source As String \* 1, target As String \* 1

TxtOutput.Text = TxtOutput.Text & "Welcome to VB 6.0 Program developed to implement Bellman Ford Algorithm:"

TxtOutput.Text = TxtOutput.Text & vbCrLf & "......................................................................."

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in Number of Vertices/Nodes in Graph: "

noOfNodes = Val(InputBox("Read in Number of Vertices/Nodes in Graph: ", "Bellman Ford"))

TxtOutput.Text = TxtOutput.Text + Str(noOfNodes)

If (noOfNodes <= 0 Or noOfNodes > MAXNODE) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Wrong Input! Please, Read in correct digit from 1 to 10:"

Else

Call readNodes(Vertices, noOfNodes)

Call readWeight(Edges, Vertices, noOfNodes)

Call DisplayWeight(Edges, noOfNodes)

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Do you wish to find Shortest Path between two Vertices/Nodes in the Graph: "

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Press 1 -> Yes"

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Press 0 or any other digit -> No"

Ch = Val(InputBox("Press 1 -> Yes, Press 0 or any other digit -> No", "Bellman Ford"))

TxtOutput.Text = TxtOutput.Text & vbCrLf & Str(Ch)

If (Ch = 1) Then

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the Source Node/Vertex: "

source = Left(InputBox("Read in the Source Node/Vertex: ", "Bellman Ford"), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & source

TxtOutput.Text = TxtOutput.Text & vbCrLf & "Read in the Target Node/Vertex: "

target = Left(InputBox("Read in the target Node/Vertex: ", "Bellman Ford"), 1)

TxtOutput.Text = TxtOutput.Text & vbCrLf & target

Call FindShortestPath(Vertices, Edges, source, target, noOfNodes)

End If

End If

TxtOutput.Text = TxtOutput.Text & vbCrLf & "\*\*\*\*\*\* Press any Key to Continue \*\*\*\*\*\*"

CmdStart.Enabled = False

End Sub

**RBC measure on Bellman Ford Algorithm implemented with VB 6.0**

1. **Input Output Complexity (IOC)**

No of input = count (N) + count (V) + count (M) + count (E) + count (Distance) + count (predecessor) + count (S) + count (t)

= 1 + 6 + 1 + 9 + 6 + 6 + 1 + 1 = 31

* No of output = count (Distance) + count (Path) + count (cost)

= 6 + 5 + 1 = 12

* No of interface = 1 (console window)
* No of files = 1 (Data storage)

IOC = No of input + No of output + No of Interface + No of files

= 31 + 12 + 1 + 1 = 45

1. **Functional Requirement (FR)**

No of functions = (FindShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( )) = 4

No of sub-process/sub-functions ⇒ Every function Fi has zero, one or more

sub-processes such as Arithmetic, Computation, Display, etc.

Function 1 - ReadNodes ( ) has 2 (Display operations, Read operations)

Function 2 - ReadWeight ( ) has 2 (Display, Read operations)

Function 3 - DisplayWeight ( ) has 1 (Display operations)

Function 4 - BellmanFordShortestPath ( ) has 2 (Display, arithmetic addition)

= 2 + 2 + 1 + 2 = 7

FR = Number of functions \* = 4 \* 7 = 28

**3. Non functional Requirement (NFR)**

Number of Non-FR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 28 + 0 = 28

**5. Product Complexity (PC)**

PC = IOC \* RC = 45 \* 28 = 1260

**6**. **Personal Complexity Attributes (PCA)**

PCA = 0.90 (Suppose Programmer Capability = High)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((1260 \* 0.90) + 0 + 0 + 0 ) \* 1 = 1134

**3.5.4 Evaluation of Complexity of JAVA Language Implementation of Bellman Ford Shortest path Algorithm**

/\* A JAVA Program to compute the Shortest Path using Bellman Ford Algorithm

\* Class Name: BellmanFord.java

\* Project Developed by

\*/

import java.util.Scanner;

public class BellmanFord {

private final int MAXNODE = 10;

private final int TRUE = 1;

private final int FALSE = 0;

private final int INFINTY = 9999;

private int[][] Edges;

public int noOfNodes;

private char[] Vertices;

Scanner scanInput = new Scanner(System.in);

public BellmanFord(int N)

{

setNoOfNodes(N);

Edges = new int[N][N];

Vertices = new char[N];

}

private int getNoOfNodes(){

return noOfNodes;

}

private void setNoOfNodes(int Nnodes){

noOfNodes = Nnodes;

}

public void ShortestPath()

{

int ch;

if (noOfNodes <= 0 || noOfNodes > MAXNODE)

{

System.out.println("Wrong Input! Please, Read in correct digit from 1 to 10:");

}

else

{

ReadNodes(Vertices); // Read in N nodes

ReadWeight(Edges, Vertices); //Read in weight/cost of each edges

DisplayWeight(Edges); // Display Matrix of Weight

System.out.println("\n Do you wish to find Shortest Path between two Vertices/Nodes in the Graph:\n ");

System.out.println("Press 1 -> Yes \n Press 0 or any other digit -> No\n");

ch = scanInput.nextInt();

if(ch == 1)

{

System.out.print("\nRead in the Source Node/Vertex: ");

String str = scanInput.next();

char source = str.charAt(0);

System.out.print("\nRead in the Target Node/Vertex: ");

str = scanInput.next();

char target = str.charAt(0);

FindShortestPath(Vertices, Edges, source, target);

}

}

}

private void ReadNodes(char[] Nodes)

{

int i;

System.out.println("Read in the names of all Nodes/Vertices in the graph");

for(i = 0; i < Nodes.length; i++)

{

System.out.print("\nRead in the names of Node/Vertex " + (i+1) + ":\t" );

String Str = scanInput.next();

Nodes[i] = Str.charAt(0);

}

}

private void ReadWeight(int[][] W, char[] Nodes)

{

int i, j;

System.out.print("\nRead in the weights/costs of all edges between 2 Vertices in the graph:\n");

for(i = 0; i < W.length; i++)

{

System.out.println();

for (j = 0; j < W[i].length; j++)

{

System.out.print("Read in the weight/cost of edge between Node "+ Nodes[i]

+ " and Node " + Nodes[j] + ": ");

W[i][j] = scanInput.nextInt();

if (W[i][j] == 0) W[i][j] = INFINTY;

}

}

}

private void DisplayWeight(int[][] W)

{

int i, j;

System.out.print("\nThe weights/costs Table of all edges between 2 Vertices in the graph:\n");

for (i = 0; i < W.length; i++)

{

for (j = 0; j < W[i].length; j++)

{

if(W[i][j] == INFINTY) System.out.printf(" INF");

else

System.out.printf("%6d", W[i][j]);

}

System.out.print("\n");

}

}

private void FindShortestPath(char Vertices[], int Edges[][], char source, char target)

{

int souce = 0, j, u, v, taget = 0, i, w, Null = -1;

int[] distance, predecessor, S;

int found = FALSE;

/\* Dynamically Allocate Array \*/

distance = new int[Vertices.length];

predecessor = new int[Vertices.length];

S = new int[Vertices.length];

/\* Verify source \*/

for (j = 0; j < Vertices.length; j++)

{

if(source == Vertices[j])

{

souce = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ System.out.print("\nWrong Input! Source Node not found; No path found: "); return;}

found = FALSE;

/\* Verify target \*/

for(j = 0; j < Vertices.length; j++)

{

if(target == Vertices[j])

{

taget = j;

found = TRUE;

break;

}

}

if (found == FALSE)

{ System.out.print("\nWrong Input! Target Node not found; No path found: "); return; }

//Implementing Bellman Ford Algorithm

//Step 1: initialize graph

for(v = 0; v < Vertices.length; v++)

{

if(v == souce) distance[v] = 0; //distance from source to source

else

distance[v] = INFINTY; //unknown distance function from source to v

predecessor[v] = Null; //predecessor node in optimal path from source

}

//Step 2: relax edges repeatedly

for(i = 0; i < Vertices.length-1; i++)

{

for(u = 0; u < Vertices.length; u++)

{

for(v = 0; v < Vertices.length; v++)

{

w = Edges[u][v];

if((distance[u] + w) < distance[v])

{

distance[v] = distance[u] + w;

predecessor[v] = u;

}

}

}

}

//return distance[], predecessor[]

System.out.print("\nDistance table: \n") ;

for(v = 0; v < Vertices.length; v++)

{

if(distance[v] == INFINTY) System.out.print("INF\t");

else

System.out.print(distance[v]+" \t");

}

System.out.print("\nPredecessor table: \n");

for(v = 0; v < Vertices.length; v++) System.out.print(predecessor[v]+1 + " \t");

//getting the shortest path

j = 0;

u = taget;

while(predecessor[u] != Null) //construct the shortest path with a stack S

{

S[j++] = u; //Push the vertex onto the stack

u = predecessor[u]; //Traverse from target to source

}

//display shortest path

System.out.print("\n\nSuccessful, A Path Found!!! \nThe Shortest Path between " +

source + " and " + target + " is:\n " + source + "--->");

for(v = j-1; v > 0; v--) System.out.print(Vertices[S[v]]+ "--->");

System.out.print(Vertices[S[0]]+ "\n");

System.out.print("\nThe weight/cost of the shortest path between Node: " +

source + " and Node: " + target + " is " + distance[taget] + "\n");

}

}

/\* A Java Program to compute the Shortest Path using Bellman Ford Algorithm

\* This is a main Class name: BellmanFordShortestPath.java

\* Project Developed by

\*/

import java.util.Scanner;

public class BellmanFordShortestPath {

public static void main(String[] args)

{

Scanner scanInput = new Scanner(System.in);

int n;

System.out.println("Welcome to Java Program developed to implement Bellman Ford Algorithm:");

System.out.println("=================================================================\n");

System.out.print("Read in Number of Vertices/Nodes in Graph: ");

n = scanInput.nextInt();

BellmanFord bellmanFord = new BellmanFord(n);

bellmanFord.ShortestPath();

}

}

**RBC measure on Bellman Ford Algorithm implemented with JAVA**

1. **Input Output Complexity (IOC)**

No of input = count (N) + count (V) + count (M) + count (E) + count (Distance) + count (predecessor) + count (S) + count (t)

= 1 + 6 + 1 + 9 + 6 + 6 + 1 + 1 = 31

* No of output = count (Distance) + count (Path) + count (cost)

= 6 + 5 + 1 = 12

* No of interface = 1 (console window)
* No of files = 1 (Data storage)

IOC = No of input + No of output + No of Interface + No of files

= 31 + 12 + 1 + 1 = 45

1. **Functional Requirement (FR)**

No of functions = (BellManFordShortestPath ( ), SetNoOfNodes ( ), GetNoOfNodes ( ), ShortestPath ( ), ReadNodes ( ), ReadWeight ( ), DisplayWeight ( ), FindShortestPath ( )) = 8

No of sub-process/sub-functions ⇒ Every function Fi has zero, one or more sub-processes such as Arithmetic, Computation, Display, etc.

Function 1 - ShortestPath ( ) has 3 (Display operations, Read operations, Type-Casting operations)

Function 2 - ReadNodes ( ) has 3 (Display operations, Read operations, Type-Casting operations)

Function 3 - ReadWeight ( ) has 3 (Display, Read, Type-casting operations)

Function 4 - DisplayWeight ( ) has 1 (Display operations)

Function 5 - FindShortestPath ( ) has 2 (Display, arithmetic addition)

= 3 + 3 + 3 + 1 + 2 = 12

FR = Number of functions \* = 8 \* 12 = 96

**3. Non functional Requirement (NFR)**

Number of Non-FR = 0 (no quality attribute)

**4. Requirement Complexity (RC)**

RC = FR + NFR = 96 + 0 = 96

**5. Product Complexity (PC)**

PC = IOC \* RC = 45 \* 96 = 4320

**6**. **Personal Complexity Attributes (PCA)**

PCA = 0.90 (Suppose Programmer Capability = High)

**7. Design Constraints Imposed (DCI)**

Number of Constraints = 00 (No directives)

DCI = 0

**8. Interface Complexity (IFC)**

IFC = 0 ( no External Interface required)

**9. Users/Location Complexity (ULC)**

Number of user = 1

Number of location = 1

ULC = No of user \* No of location = 1 \* 1 = 1

**10. System Feature complexity**

SFC = 0 (no specific features required)

**Requirement Based Complexity (RBC)**

RBC = ((PC \* PCA) + DCI + IFC + SFC) \* ULC

= ((4320 \* 0.90) + 0 + 0 + 0 ) \* 1 = 3888

**CHAPTER FOUR**

**4.0 RESULTS AND DISCUSSION**

**4.1 RESULTS**

Having implemented the three shortest path (Dijkstra, Floyd-Warshall, and Bellman-Ford) algorithms by C#, C++, Java, and Visual Basic programming languages, the following results were obtained:

**Table 4.1: Requirement Based Complexity (RBC) Measure values for Dijkstra Algorithm implemented using C#, C++, VB, and Java.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Implementation Languages | | | |
|  | C# | C++ | VB | Java |
| Algorithm | Dijkstra | Dijkstra | Dijkstra | Dijkstra |
| RBC Values | 4043.52 | 1555.2 | 1008 | 3456 |

**Table 4.2: Requirement Based Complexity (RBC) Measure values for Floyd-Warshall Algorithm implemented using C#, C++, VB, and Java.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Implementation Languages | | | |
|  | C# | C++ | VB | Java |
| Algorithm | Floyd-Warshall | Floyd-Warshall | Floyd-Warshall | Floyd-Warshall |
| RBC Values | 12542.4 | 5065.2 | 3618 | 10720 |

**Table 4.3: Requirement Based Complexity (RBC) Measure values for Bellman-Ford Algorithm implemented using C#, C++, VB, and Java.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Implementation Languages | | | |
|  | C# | C++ | VB | Java |
| Algorithm | Bellman-Ford | Bellman-Ford | Bellman-Ford | Bellman-Ford |
| RBC Values | 5054.4 | 1822.5 | 1134 | 3888 |

**Table 4.4: Requirement Based Complexity (RBC) Measure values for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithms implemented using C# language.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Algorithms | | |
|  | Dijkstra | Floyd-Warshall | Bellman-Ford |
| Implementation Language | C# | C# | C# |
| RBC Values | 4043.52 | 12542.4 | 5054.4 |

**Table 4.5: Requirement Based Complexity (RBC) Measure values for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithms implemented using C++ language.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Algorithms | | |
|  | Dijkstra | Floyd-Warshall | Bellman-Ford |
| Implementation Language | C++ | C++ | C++ |
| RBC Values | 1555.2 | 5065.2 | 1822.5 |

**Table 4.6: Requirement Based Complexity (RBC) Measure values for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithms implemented using VB language.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Algorithms | | |
|  | Dijkstra | Floyd-Warshall | Bellman-Ford |
| Implementation Language | VB | VB | VB |
| RBC Values | 1008 | 3618 | 1134 |

**Table 4.7: Requirement Based Complexity (RBC) Measure values for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithms implemented using Java language.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Algorithms | | |
|  | Dijkstra | Floyd-Warshall | Bellman-Ford |
| Implementation Language | JAVA | JAVA | JAVA |
| RBC Values | 3456 | 10720 | 3888 |

**Fig. 4.1: Graph of Requirement Based Complexity (RBC) measure values for Dijkstra Algorithm implemented using C#, C++, VB, and JAVA.**

The graph in figure 4.1 above shows values of Requirement Based Complexity (RBC) for Dijkstra Algorithm implemented using C#, C++, VB, and JAVA languages. It is explicitly shown that dijkstra algorithm has the highest Requirement Based Complexity (RBC) value when implemented using C# language and has the least Requirement Based Complexity (RBC) value when implemented using VB language. This means that dijkstra algorithm is better implemented using C# than any other three (3) languages in terms of RBC measure.

**Fig. 4.2: Graph of Requirement Based Complexity (RBC) measure values for Floyd-Warshall Algorithm implemented using C#, C++, VB, and JAVA.**

The graph in figure 4.2 above shows values of Requirement Based Complexity (RBC) for Floyd-Warshall Algorithm implemented using C#, C++, VB, and JAVA languages. It is explicitly shown that Floyd-Warshall algorithm has the highest Requirement Based Complexity (RBC) value when implemented using C# language and has the least Requirement Based Complexity (RBC) value when implemented using VB language. This means that Floyd-Warshall algorithm is better implemented using C# than any other three (3) languages in terms of RBC measure.

**Fig. 4.3: Graph of Requirement Based Complexity (RBC) measure values for Bellman-Ford Algorithm implemented using C#, C++, VB, and JAVA.**

The graph in figure 4.2 above shows values of Requirement Based Complexity (RBC) for Bellman-Ford Algorithm implemented using C#, C++, VB, and JAVA languages. It is explicitly shown that Bellman-Ford algorithm has the highest Requirement Based Complexity (RBC) value when implemented using C# language and has the least Requirement Based Complexity (RBC) value when implemented using VB language. This means that Bellman-Ford algorithm is better implemented using C# than any other three (3) languages in terms of RBC measure.

**Fig. 4.4: Graph of Requirement Based Complexity (RBC) measure values for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithm implemented using C# Language.**

The graph in figure 4.2 above shows values of Requirement Based Complexity (RBC) for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithm implemented using C# language. It is explicitly shown that Floyd-Warshall algorithm has the highest Requirement Based Complexity (RBC) value when implemented using C# language and Dijkstra algorithm has the least Requirement Based Complexity (RBC) value when implemented using C# language. This means that Floyd-Warshall algorithm is better implemented using C# language than the other two (2) algorithms in terms of RBC measure.

**Fig. 4.5: Graph of Requirement Based Complexity (RBC) measure values for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithm implemented using C++ Language.**

The graph in figure 4.2 above shows values of Requirement Based Complexity (RBC) for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithm implemented using C++ language. It is explicitly shown that Floyd-Warshall algorithm has the highest Requirement Based Complexity (RBC) value when implemented using C++ language and Dijkstra algorithm has the least Requirement Based Complexity (RBC) value when implemented using C++ language. This means that Floyd-Warshall algorithm is better implemented using C++ language than the other two (2) algorithms in terms of RBC measure.

**Fig. 4.6: Graph of Requirement Based Complexity (RBC) measure values for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithm implemented using VB Language.**

The graph in figure 4.2 above shows values of Requirement Based Complexity (RBC) for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithm implemented using VB language. It is explicitly shown that Floyd-Warshall algorithm has the highest Requirement Based Complexity (RBC) value when implemented using VB language and Dijkstra algorithm has the least Requirement Based Complexity (RBC) value when implemented using VB language. This means that Floyd-Warshall algorithm is better implemented using VB language than the other two (2) algorithms in terms of RBC measure.

**Fig. 4.7: Graph of Requirement Based Complexity (RBC) measure values for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithm implemented using JAVA Language.**

The graph in figure 4.2 above shows values of Requirement Based Complexity (RBC) for Dijkstra, Floyd-Warshall, and Bellman-Ford Algorithm implemented using JAVA language. It is explicitly shown that Floyd-Warshall algorithm has the highest Requirement Based Complexity (RBC) value when implemented using JAVA language and Dijkstra algorithm has the least Requirement Based Complexity (RBC) value when implemented using JAVA language. This means that Floyd-Warshall algorithm is better implemented using JAVA language than the other two (2) algorithms in terms of RBC measure.

**4.2 DISCUSSION**

In table 4.1 and the graph in figure 4.1 above, it was clearly shown that Dijkstra algorithm implemented in C# language has the highest Requirement Based Complexity (RBC) value and has the highest functionality to be performed. This means that, out of the four(4) implementation languages (C#, C++, VB, and Java), C# language is the most efficient language for implementing Dijkstra algorithm and VB language is the least efficient language for implementing Dijkstra algorithm.

In table 4.2 and the graph in figure 4.2 above, it was clearly shown that Floyd-Warshall algorithm implemented in C# language has the highest Requirement Based Complexity (RBC) value and has the highest functionality to be performed. This means that, out of the four(4) implementation languages (C#, C++, VB, and Java), C# language is the most efficient language for implementing Floyd-Warshall algorithm and VB language is the least efficient language for implementing Floyd-Warshall algorithm.

In table 4.3 and the graph in figure 4.3 above, it was clearly shown that Bellman-Ford algorithm implemented in C# language has the highest Requirement Based Complexity (RBC) value and has the highest functionality to be performed. This means that, out of the four(4) implementation languages (C#, C++, VB, and Java), C# language is the most efficient language for implementing Bellman-Ford algorithm and VB language is the least efficient language for implementing Bellman-Ford algorithm.

In table 4.4 – 4.7 and the graph in figure 4.4 – 4.7 above, it was clearly shown that Floyd-Warshall algorithm implemented in C#, C++, VB, and JAVA languages has the highest Requirement Based Complexity (RBC) value and has the highest functionality to be performed. This means that, out of the three(3) shortest path algorithms (Dijkstra, Floyd-Warshall, and Bellman-Ford), Floyd-Warshall is the most efficient shortest path algorithm.