Chapter 4

Algorithms for Generating Star \( S_n \) and Path \( P_n \) of Graphs using BFS

4.1. Introduction:

4.1.1 Walks, trails, paths:

If \( u \) and \( v \) are two vertices in a graph \( G \), a \( u \to v \) walk is an alternating sequence of vertices and edges starting with \( u \) and ending at \( v \). Consecutive vertices and edges are incident. For the graph [5] in Figure 4.1, an example of a walk is an \( a \to e \) walk: \( a, h, c, b, e \). In other words, we start at vertex \( a \) and travel to vertex \( b \). From \( b \), we go to \( c \) and then back to \( b \) again. Then we end our journey at \( e \). Notice that consecutive vertices in a walk are adjacent to each other. One can think of vertices as destinations and edges as footpaths, say. We are allowed to have repeated vertices and edges in a walk. The number of edges in a walk is called its length. For instance, the walk \( a, b, c, b, e \) has length 4.

A trail is a walk with no repeating edges. For example, the \( a \to b \) walk \( a, b, c, d, f, g, b \) in Figure 4.1 is a trail. It does not contain any repeated edges, but it contains one repeated vertex, i.e. \( b \). Nothing in the definition of a trail restricts a trail from having repeated vertices. Where the start and end vertices of a trail are the same, we say that the trail is a circuit, otherwise known as a closed trail. Thus the walk \( a, b, e, a \) is a circuit. A walk with no repeating vertices is called a path. Without any repeating vertices, a path cannot have repeating edges; hence a path is also a trail. A path whose start and end vertices are the same is called a cycle. For example, the walk \( a, b, e, a \) in Figure 4.1 is a path and a cycle.

![Figure 4.1: Walking along a graph](image-url)
4.1.2 Cycle Graphs:

A cycle graph $C_n$, sometimes simply known as an $n$-cycle, is a graph on $n$-nodes containing a single cycle through all nodes. Special cases include $C_3$ (the triangle graph), $C_4$ (the square graph, also isomorphic to the grid graph $C_{22}$), $C_6$ (isomorphic to the bipartite Kneser graph $K(3,1)$), and $C_8$ (isomorphic to the 2-Hadamard graph). The $2n$-cycle graph is isomorphic to the Haar graph.

![Figure 4.2: Cycle Graph](image)

4.1.3 Path Graphs:

The path $P_n$ is a tree with two nodes of vertex degree 1, and the other $n - 2$ nodes of vertex degree 2. A path graph is a graph that can be drawn so that all of its vertices and edges lie on a single straight line.

The path graph $P_1$ is known as the singleton graph and is equivalent to the complete graph $K_1$ and the star graph $S_1$. Path graphs $P_n$ are always graceful for $n > 4$.

![Figure 4.3: Path Graph](image)

4.1.4 Complete graph:

A complete graph is a graph in which each pair of graph vertices is connected by an edge. The complete graph with $n$ graph vertices is denoted $K_n$ and has $\binom{n}{2}$ connections.
The complete graph on 0 nodes is a trivial graph known as the null graph, while the complete graph on 1 node is a trivial graph known as the singleton graph. $K_3$ is the cycle graph $C_3$, as well as the odd graph $O_2$. $K_4$ is the tetrahedral graph, as well as the wheel graph $W_4$, and is also a planar graph. $K_5$ is nonplanar, and is sometimes known as the pentatope graph or Kuratowski graph. Conway and Gordon (1983) proved that every embedding of $K_6$ is intrinsically linked with at least one pair of linked triangles, and $K_6$ is also a Cayley graph. Conway and Gordon (1983) also showed that any embedding of $K_7$ contains a knotted Hamiltonian cycle. The complete graph $K_n$ is the line graph of the star graph $S_{n+1}$.

4.1.5 Star Graph:

The star graph $S_n$ of order $n$, sometimes simply known as an "n-star", is a tree on $n$-nodes with one node having vertex degree $n - 1$ and the other $n - 1$ having vertex degree 1. The star graph $S_n$ is therefore isomorphic to the complete bipartite graph $K_{1,n-1}$.
4.1.6 Wheel Graph:

A wheel graph \( W_n \) of order \( n \), sometimes simply called an \( n \)-wheel, is a graph that contains a cycle of order \( n - 1 \), and for which every graph vertex in the cycle is connected to one other graph vertex (which is known as the hub). The edges of a wheel which include the hub are called spokes. The wheel \( W_n \) can be defined as the graph \( K_1 + C_{n-1} \), where \( K_1 \) is the singleton graph and \( C_1 \) is the cycle graph. Note that there are two conventions for the indexing for wheel graphs, with some authors, adopting the convention that \( W_n \) denotes the wheel graph on \( n + 1 \) nodes.

4.1.7 Windmill Graph:

The windmill graph \( D_n^{(m)} \) is the graph obtained by taking \( m \) copies of the complete graph \( K_n \) with a vertex in common. The case \( n = 3 \) therefore corresponds to the Dutch windmill graph \( D_n^{(m)} \).
4.1.8 Diamond Graph:

The diamond graph is the simple graph on 4 nodes and 5 edges illustrated below.

\[ \text{Figure 4.8: Diamond} \]

4.1.9 Paw Graph:

The paw graph is the 3-pan graph, which is also isomorphic to the (3,1)-tadpole graph.

\[ \text{Figure 4.9: Paw} \]
4.1.10 Gem Graph:

The gem graph is the fan graph $F_{4,1}$ illustrated below.

![Gem Graph](image)

Figure 4.10: Gem

4.1.11 Dart Graph:

The dart graph is the 5-vertex graph illustrated below.

![Dart Graph](image)

Figure 4.11: Dart

4.1.12 Tetrahedral:

The tetrahedral graph is the Platonic graph that is the unique polyhedral graph on four nodes which is also the complete graph $K_4$ and therefore also the wheel graph $W_4$.

![Tetrahedral Graph](image)

Figure 4.12: Tetrahedral
4.1.13 **Concatenation:**

The act of linking together as in a series or chain.

4.1.14 **Concatenation Graph $C_k P_l$:**

A Cycle and Path class of graph linked together as in a series $C_k P_l$ as shown below.

![Concatenation Graph](image)

Figure 4.13: Concatenation Graph $C_k P_l$

4.1.15 **Breadth First Search Algorithm (BFS):**

Breadth first search is another useful tool in many graph algorithms. The BFS visits systematically the vertices of graph or digraph, beginning at some vertex $r$ of $G$ (also called a root). The root is the first active vertex. At any stage during the search, all the vertices adjacent from the current active vertex are scanned for vertices that have not yet been visited that are "broad" search performed for unvisited vertices. Each time a vertex is visited for the first time, it is labeled (according to some rule that depends on the goal to be achieved) and added back to the queue. Note that, in this search a queue is used rather than a stack. The current active vertex is the one at the front of the queue. As soon as its neighbors have been visited, it is deleted from the queue. If the queue is empty and some vertices of the graph or diagraph have not
yet been visited, we select any unvisited vertex, assign it a label and add it to the queue. When all the vertices of the graph have been visited, the search is complete.

Applications:

1. Finds a tree.
2. Tests cross edge connectivity.
3. Find paths.
4. Find a cyclicity.

We now describe one kind of labeling of the vertices of a graph during a BFS. Assume that $G$ is a graph represented by its adjacency list. Initially, all vertices of $G$ are labeled 0. We begin by assigning $r$ the label 1 and placing $r$ on a queue $Q$. At the next step, we delete $r$ from $Q$ and scan its adjacency vertices (if such vertices exit) sequentially in the order in which they appear on the adjacency list of $r$. The first vertex that appears on the adjacency list for $r$ is assigned the next available, namely 2, and this vertex is then added to (back of) $Q$. We continue to label the vertices adjacent to $r$ and add them to $Q$ until the last vertex adjacent with $r$ is labeled $\text{deg } r + 1$ and is added to $Q$. We then delete the next vertex from (the front of) $Q$, say $w$, and scan its adjacent vertices in the order in which they appear on the adjacency list of $w$. If a vertex adjacent with $w$ still has 0, then we assign it the next available labels and add it to $Q$; otherwise, we do not change its label. We continue in this manner until $Q$ is empty. If all the vertices of $G$ are labeled with a positive integer, we stop. This algorithm actually determines a spanning tree of $G$, called BFS.

Queue

A queue is a list of elements which supports the following operations

- enqueue: Adds an element to the end of the list
- dequeue: Removes an element from the front of the list

Elements are extracted in first-in first-out (FIFO) order, i.e., elements are picked in the order in which they were inserted.
Algorithm BFS (v):

// visits all the vertices connected to root v by a path
// assigns them the numbers in the order they are visited
// via global variable count

count ← count + 1; mark v with count and initialize a queue with v.

while the queue is not empty do

for each vertex w in V adjacent to the front vertex do

if w is marked with 0

count ← count + 1; mark w with count
add w to the queue

remove the front vertex from the queue

Figure 4.14

Constructing a BFS Tree for above figure:

Queue
V
V1 V2 V4 V6 V7 V8
4.2 Proposed work:

Our aim is to find the small class of graphs which can result in BFS tree which is star graph called BFS Star $S_n$ and BFS tree which is path called BFS Path $P_n$, after application of BFS. i.e. $BFST(G) = Star \ S_n$ and $BFST(G) = Path \ P_n$.

Some graphs will result in Star (G) and Path (G) by direct application of BFS where some need modifications in the algorithm. The BFS algorithm starts with a root vertex called start vertex. The resulted output tree structure will be in the form of Star $S_n$ Structure or in Path $P_n$ Structure.

In this chapter we are dealing with some small basic graphs and their behavior on applying BFS. Some of the graph classes results in $BFST(G) = Star \ S_n$ and $BFST(G) = P_n$ after applying some conditions and by specifying start vertex and modifying the BFS algorithm accordingly. We want to generate a Star $S_n$ and Path $P_n$ by applying BFS on the given graph. We first show some class of graphs which results in $BFST(G) = Star \ S_n$ and then will show some class of graphs which results in $BFST(G) = P_n$ by finding the start vertex by modifying BFS algorithm. To elaborate the concept let us consider the following example a Paw Graph which results both $BFST(G) = Star \ S_n$ and $BFST(G) = P_n$. 

![BFST (G) with start vertex V1]
\( \text{BFST}(G) = \text{Star } S_n: \)

For the given graph, in Fig.4.15 if we apply BFS with vertex \( V_2 \) as start vertex, then we get \( \text{BFST}(G) \)

\[ \text{Queue} \]
\[ V_2 \]
\[ V_1, V_3, V_4 \]

\( \text{BFST}(G) \) with start vertex \( V_2 \)

The structure of \( \text{BFST}(G) \) is in the form of \( \text{Star } S_4 \).

If we take \( V_1 \) also as start vertex we get a \( \text{BFST}(G) = \text{Star } S_n \)

i.e.

\[ \text{Queue} \]
\[ V_1 \]
\[ V_2 \]
\[ V_3, V_4 \]

\( \text{BFST}(G) \) with start vertex \( V_1 \)
The structure of $BFST(G)$ is in the form of Star $S_4$.

$BFST(G) = P_n$:

For the given graph in above fig.4.15, if we apply BFS with vertex $V_3$ as a start vertex, then we get $BFST(G)$ as described below

Therefore the $BFST(G)$ is in the form of Path $P_4$

In the same way if we take $V_4$ as a start vertex then we get result as $BFST(G) = P_4$.

This example clearly shows that same graph has different $BFST(G)$ structures, taking different start vertex which is Star $S_n$ as well as Path $P_n$. Just we have to identify the start vertex which is done by modifying BFS algorithm.

Another class of graph i.e. Diamond graph which gives both $BFST(G) = Star(G)$ and $BFST(G) = P_n$. Figure: 4.16
Consider $V_1$ as a start vertex

Therefore $BFST(G) = P_n$

i.e.

In the same way if we take $V_4$ as a start vertex we will get Path $P_4$.

$BFST(G) = Star(G)$

Consider $V_2$ as a root vertex

Queue: $V_2$
Therefore BFST(G) = Star S₃.
In the same way if we take V₃ as a start vertex we will get Star S₄.

4.2.1 Graph Classes that gives BFS Star Sₙ, by applying BFS without any conditions:

1. Star graph:

   When G is star, then applying BFS with taking any vertex as a start vertex it gives star itself. Here each vertex is a start vertex.

   For e.g.: Consider Star (G) K₁₆ as shown below:

   ![](image)

   Figure 4.17

   Applying BFS to above fig 4.17, let V₂ be start vertex:

   ![](image)
Therefore the \((K_{1,n}) = BFS\ Star K_{1,n}\).

2. Complete graph:

In complete graph also each vertex is start vertex. When \(G\) is complete, then applying BFS with taking any vertex as start vertex it gives \(BFS\ Star\ S_n\).

For e.g.: Consider \(K_4\) as shown below:

![Figure 4.18](image)

Applying BFS to above fig 4.18, let \(V_2\) be start vertex:

![Queue](image)

Therefore \(BFS T(K_n) = BFS\ Star\ S_4\).

3. Tetrahedral Graph:

When \(G\) is Tetrahedral, then applying BFS with taking any vertex as start vertex it gives \(BFS\ Star\ S_n\). Here also every vertex is start vertex.

For e.g.: Consider tetrahedral graph as shown below:

![Figure 4.19](image)

Applying BFS to above fig 4.19, let \(V_4\) be start vertex:
Therefore \( BFST(\text{tetrahedral}) = BFS \ Star \ S_n \).

4.2.2 Graph classes that give BFS Star \( S_n \) by specifying a Start vertex:

Algorithm BFS Star \( S_n \) by BFS (G)

// I/P: Graph \( G = (V, E) \).

// O/P: Star Graph \( S_n \) in the order they have been visited by BFS traversal.

Step 1: Find the degree of vertices and find the max deg \( (\Delta(G)) \) of a Graph.

Step 2: Let the start vertex ‘S’ be the vertex with max deg \( (\Delta(G)) \).

Step 3: Apply BFS on given graph with ‘S’ as start vertex.

Step 4: We get the output as a BFS Star \( S_n \).

In this section we will study about how the start vertex ‘S’ of given graph is obtained so that vertices are visited in a specific order from start vertex. For some class of graphs we can obtain \( BFST(G) = BFSStar \ S_n \) by specifying start vertex and applying BFS algorithm accordingly. In the same way we show certain class of graph which results \( BFST(G) = BFSStarS_n \).

1. Wheel Graph:

Now let us consider wheel graph \( W_6 \) as shown in below figure 4.20. by applying the above algorithm we get
According to our modified algorithm in the above figure first we find the degree of each vertex. There are total 7 vertices. The degree of each vertex is:

- $V_1 \rightarrow 3$
- $V_2 \rightarrow 3$
- $V_3 \rightarrow 3$
- $V_4 \rightarrow 3$
- $V_5 \rightarrow 3$
- $V_6 \rightarrow 3$
- $V_7 \rightarrow 6$

Therefore $V_7$ is a vertex with max degree $\Delta(G) \rightarrow 6$. Let $V_7$ be a start vertex.

Applying BFS to above fig 4.20, let $V_7$ be start vertex:

Queue

$V_7$

$V_1V_2V_3V_4V_5V_6$

Therefore $BFST$ Wheel graph $W_n = BFS$ Star $S_n$. 
In the same way we show certain class of graphs which results

\[ BFST(G) = BFSStarS_n \]

2. Windmill Graph:

Consider \( W_3 \) graph as shown below

![Figure 4.21](image)

The degree of each vertex is:

- \( V_1 \) \( \rightarrow \) 2
- \( V_2 \) \( \rightarrow \) 2
- \( V_3 \) \( \rightarrow \) 2
- \( V_4 \) \( \rightarrow \) 2
- \( V_5 \) \( \rightarrow \) 2
- \( V_6 \) \( \rightarrow \) 2
- \( V_7 \) \( \rightarrow \) 6

Therefore \( V_7 \) is a vertex with max degree \( \Delta(G) \) \( \rightarrow \) 6. Let \( V_7 \) be a start vertex.

Applying \( BFS \) to above fig4.21, let \( V_7 \) be start vertex:

Queue

- \( V_7 \)

119
Therefore BFST Windmill graph $W_n^m = BFS Star S_n$.

3. Dart Graph:
   Consider Dart graph as shown below:

The degree of each vertex is:

\[ V_1 \rightarrow 4 \]
\[ V_2 \rightarrow 1 \]
\[ V_3 \rightarrow 2 \]
\[ V_4 \rightarrow 3 \]
\[ V_5 \rightarrow 2 \]

In this graph we have $\Delta(G) \rightarrow 4$ and $\delta(G) \rightarrow 1$ and degree 2,3 also.

Therefore $V_1$ is a vertex with max degree $\Delta(G) \rightarrow 4$. Let $V_1$ be a start vertex.

Applying BFS to above fig 4.22, let $V_1$ be start vertex:

Queue

\[ \bullet V_1 \]

\[ V_1 \]
Therefore $BFST$ Dart graph $= BFS$ Star $S_n$.

4. **Gem Graph:**

Consider Gem graph as shown below

![Gem Graph Diagram](image)

Figure 4.23

The degree of each vertex is:

- $V_1 \rightarrow 2$
- $V_2 \rightarrow 4$
- $V_3 \rightarrow 2$
- $V_4 \rightarrow 3$
- $V_5 \rightarrow 3$

In this graph we have $\Delta(G) \rightarrow 4$ and $\delta(G) \rightarrow 2$ and degree 3 also

Therefore $V_2$ is a vertex with max degree $\Delta(G) \rightarrow 4$. Let $V_2$ be a start vertex.

Applying $BFS$ to above fig 4.23, let $V_1$ be start vertex:
Therefore $BFST$ Gem graph = $BFS$ Star $S_n$.

These are the class of graph which gives $BFST(G) = BFS$ Star $S_n$.

4.2.3 Flow Chart of Algorithm BFS Star $S_n$ by BFS (G):

```
Start

Graph $G(V, E)$

initialize adjacent matrix

FDM()

BFS (G)

BFS Star $S_n$

Stop
```

Fig.2.24 Flow Chart of Algorithm BFS Star $S_n$ by BFS (G)
4.2.4 Program:

```cpp
#include<iostream.h>
#include<conio.h>
const int MAX=10;
int visited[MAX],count=0;  // Linked list for nodes

struct list
{
    int v;                //vertex
    int w;                //weight
    // int count,temp;
    struct list *link;
};                        //adjacency list class

class adjlist
{
    protected:
        list *a[MAX];    //array of pointers
        int n;          //no of vertices

    public:
        adjlist();
        int readadj();
        void printadj();
        int getvertices();
        void bfs(int);
        void qinsert(int);
        int qdelete();
        list *front,*rear;

};

adjlist :: adjlist()
{
    n=10;
    for(int i=1;i<=n;i++)
        a[i]=0;
}
```
int adjlist::readadj()
{
    list *temp,*q;
    int vertex,weight,t=0;
    char choice;
    cout<<"Enter no of vertices:";
    cin>>n;
    for (int i=1;i<=n;i++)
    {
        do
        {
            cout<<"enter adjacent vertex and weight for:"<<i<<":";
            cin>>vertex>>weight;
            if(vertex==0)break;
            temp=new list;
            temp->v=vertex;
            temp->w=weight;
            temp->link=0;
            a[i]=temp;
            else
            {
                q=a[i];
                while(q->link)
                {
                    q->link;
                    q->link=temp;
                }
                cout<<"you want to continue(y/n)?";
                cin>>choice;
                if(choice=='y')
                {
                    ++count;
                }
            }
        }
        if(count>t)
        124
void adjlist::qinsert(int x)
{
    list *q;
    q=new list;
    // insert logic here
}

void adjlist::bfs(int s)
{
    list *q;
    front=rear=0;
    visited[s]=1;
    while(1)
    {
        q=a[s];
        while(q)
        {
            if(!visited[q->v])
            {
                qinsert(q->v);
                cout<<q->v<<endl;
                visited[q->v]=1;
            }
            q=q->link;
        }
        if(!front) return;
        s=qdelete();
    }
}

{    
    t=count;
    count=0;

} while(choice=='y' || choice=='Y');

return (t);
```cpp
q->v=x;q->link=0;
if(!front)
{
    front=rear=q;
    return;
}
rear->link=q;rear=q;
}
int adjlist::qdelete()
{
    int x;
    if (front) x=front->v;
    if(front==rear)
        front=rear=0;
    else
        front=front->link;
    return x;
}
int adjlist::getvertices()
{return(n);} 
void adjlist::printadj()
{
    list*q;
    cout<<"\n adjacent list output:\n";
    for (int i=1;i<=n;i++)
    {
        q=a[i];
        for(;q;q=q->link)
            cout<<"<"<<i<<","<<q->v<<""<<q->w<<endl;
    }
}
void main()
{
    clrscr();
 ```
adjlist g;
int t;
t=g.readadj();
g.printadj();
cout<<"\n\n the given graph maximum of degree vertex is:"<<t;
cout<<"\n in vertix reachable from:"<<t<<"to\n";
g.bfs(t);
cout<<"\n";
getch();
}

Output:
Enter no of vertices: 4
Enter adjacent vertex and weight for: 1:2 0
You want to continue(y/n)? y
Enter adjacent vertex and weight for: 1:3 0
You want to continue(y/n)? n
Enter adjacent vertex and weight for: 2:1 0
You want to continue(y/n)? y
Enter adjacent vertex and weight for: 2:4 0
You want to continue(y/n)? n
Enter adjacent vertex and weight for: 3:1 0
You want to continue(y/n)? y
Enter adjacent vertex and weight for: 3:4 0
You want to continue(y/n)? n
Enter adjacent vertex and weight for: 4:2 0
You want to continue(y/n)? y
Enter adjacent vertex and weight for: 4 3 0

You want to continue (y/n)? n

Adjacent list output:

<1,2>0
<1,3>0
<2,1>0
<2,4>0
<3,1>0
<3,4>0
<4,2>0
<4,3>0

The given graph maximum of degree vertex is: 2

Vertex reachable from: 2 to

1
4
3
4.2.5 Graph Classes that gives BFS Path $P_n$, by applying BFS without any conditions:

1. Cycle:

When $G$ is cycle $C_n$, then by applying the BFS simply gives a $P_n$. Of course the cycle results in $P_n$ just by removing the edge also. But here we are studying with respect to BFS. In cycle every vertex is a start vertex. For example consider $C_6$ as shown below

![Diagram of C6 graph](image)

Figure 4.25

Applying BFS to above fig 4.25, let $V_1$ be start vertex:

```
Queue
· $V_1$

$V_1$

$V_6$
$V_2$

$V_1$

$V_6$
$V_2$

$V_5$
$V_3$

$V_5$
$V_3$
```

129
Therefore $BFST(G) = BFS P_n$

2. Path:

By applying BFS on any path graph $P_n$, we get $BFS Path P_n$ it as a result. Here each vertex is a start vertex. For example consider $P_6$ as shown below

Figure 4.26

Therefore $BFST(G) = BFS P_n$.

4.2.6 Graph classes that give BFS Path ($P_n$) by specifying a Start vertex:

Algorithm BFS Path $P_n$ by BFS ($G$)

// I/P: Graph $G = (V, E)$.

// O/P: Path Graph $P_n$ in the order they have been visited by BFS traversal.

Step 1: Find the diameter of cycle $C_k$ which is $k/2$.

Step 2: Find a vertex 'v' which is at a dist = diam $C_k$ or $k/2$ from the concatenated vertex of $C_k P_1$.

Step 3: Select v as a start vertex.
Step 4: Apply BFS using 's' as start vertex.

Step 5: We get output $P_{(i+k-1)}$.

In this section we will study about how the start vertex 'v' of given graph is obtained so that vertices are visited in a specific order from start vertex. For concatenated graph $C_k . P_t$ class of graphs we can obtain $BFST(G) = BFSPath P_n$ by specifying start vertex and applying BFS algorithm accordingly. Here if cycle is even we get one start vertex, if odd we get two start vertex we can take any one of them as a start vertex.

Consider a graph $C_6 . P_4$ as shown below

![Figure 4.27: C_6 . P_4](image)

According to our modified algorithm applying on above graph, in which first step we find the diameter of $C_6$

$$Diam \ C_k = \frac{k}{2}$$

$$Diam \ C_6 = \frac{6}{2} = 3$$

Therefore $Diam \ C_6 = 3$

Here concatenated vertex of $C_k . P_t$ of given graph is $V_4$
In second step we find vertex \( v \) from vertex \( V_4 \) at distance 3. i.e. \( dslt = diam \, C_k \).
Therefore here the vertex is \( V_7 \). In third step here we take \( V_7 \) as a start vertex any one of them. In step four we apply BFS to start vertex, consider \( V_6 \) as a start vertex.

Applying BFS to above fig 4.27, let \( V_6 \) be start vertex:

\begin{itemize}
  \item \( V_7 \)
  \item \( V_6 \)
  \item \( V_5 \)
  \item \( V_4 \)
  \item \( V_3 \)
\end{itemize}
Therefore $BFST(G) = BFS P_n$.
4.2.7 Flow Chart of Algorithm BFS Path $P_n$ by BFS (G)

Start

Graph $G_0, P_i$

initialize adjacent matrix

Find the cycle

Find the no. of vertices in cycle

Diameter $= k/2$

assign start vertex as $s = c + d$

from $s$ start BFS

BFS (G)

BFS Path $P_n$

Stop

$c = $ concatenated vertex.

d = diameter of the cycle.
4.2.8 Program:

```c
#include<alloc.h>
#define TRUE 1
#define FALSE 0
#define MAX 8
struct node
{
    int data;
    struct node *next;
}
int visited[MAX];
int q[8];
int front, rear;
void bfs ( int, struct node ** );
struct node * getnode_write ( int );
void addqueue ( int );
int deletequeue();
int isempty();
void del ( struct node * );
void samp();
void main()
{
    struct node *arr[MAX];
    struct node *v1, *v2, *v3, *v4;
    int temp[1][8],max=0,m=0,y=0,ij;
    int a[10][10],n,b=3;
    clrscr();
    printf("enter the vertices\n");
    scanf("%d",&n);
    if(n<9)
        printf("Invalid Data");
    else
    {
        printf("Enter the arry elments\n");
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                scanf("%d",&a[i][j]);
        for(i=0;i<n;i++)
        {
            m=0;
            for(j=0;j<n;j++)
                if(a[i][j]==1)
```
m++;  
if(max<m)  
{    max=m;    
y=i;  
}  
}
printf("%d node has maximum degree\n",y+l);
printf("No of edges connected are %d\n",max);
printf("diameter of given cycle c6 is %d\n",b);
printf("path the graph follows is shown below\n");  
samp();  
}
getch();
}
void samp()  
{
struct node *arr[MAX] ;
struct node *v1, *v2, *v3, *v4 ;
int i ;
v1 = getnode_write (2 );
arr[0] = v1 ;
v1 -> next = v2 = getnode_write (3 );
v2 -> next = NULL ;
v1 = getnode_write (1 );
arr[1] = v1 ;
v1 -> next = v2 = getnode_write (4 );
v2 -> next = v3 = getnode_write (5 );
v3 -> next = NULL ;
v1 = getnode_write (1 );
arr[2] = v1 ;
v1 -> next = v2 = getnode_write (6 );
v2 -> next = v3 = getnode_write (7 );
v3 -> next = NULL ;
v1 = getnode_write (2 );
arr[3] = v1 ;
v1 -> next = v2 = getnode_write (8 );
v2 -> next = NULL ;
v1 = getnode_write (2 );
arr[4] = v1 ;
v1 -> next = v2 = getnode_write (8 );
v2 -> next = NULL ;
v1 = getnode_write (3 );
arr[5] = v1 ;
v1 -> next = v2 = getnode_write (8);
v2 -> next = NULL;
v1 = getnode_write (3);
arr[6] = v1;
v1 -> next = v2 = getnode_write (8);
v2 -> next = NULL;
v1 = getnode_write (4);
arr[7] = v1;
v1 -> next = v2 = getnode_write (5);
v2 -> next = v3 = getnode_write (6);
v3 -> next = v4 = getnode_write (7);
v4 -> next = NULL;
front = rear = -1;
bfs (1, arr);
for (i = 0; i < MAX; i++)
    del (arr[i]);
//getch();

void bfs (int v, struct node **p)
{
    struct node *u;
    visited[v-1] = TRUE;
    printf ("%d	", v);
    addqueue (v);
    while (isempty() == FALSE)
    {
        v = deletequeue();
        u = *(p + v-1);
        while (u != NULL)
        {
            if (visited [u -> data -1] == FALSE)
                {
                    addqueue (u -> data);
                    visited [u -> data -1] = TRUE;
                    printf ("%d", u -> data);
                }
            u = u -> next;
        }
    }
}

struct node *getnode_write (int val)
{
    struct node *newnode;
    newnode = (struct node *) malloc (sizeof (struct node));

newnode -> data = val;
return newnode;
}

void addqueue ( int vertex )
{
    if ( rear == MAX - 1 )
    {
        printf ( "Queue Overflow.\n" ) ;
        exit( ) ;
    }
    rear++ ;
    q[rear] = vertex ;
    if ( front == -1 )
        front = 0 ;
}

int deletequeue() 
{
    int data ;
    if ( front == -1 )
    {
        printf ( "Queue Underflow.\n" ) ;
        exit( ) ;
    }
    data = q[front] ;
    if ( front == rear )
        front = rear = -1 ;
    else
    {
        front++ ;
        return data ;
    }
}

int isempty() 
{
    if ( front == -1 )
        return TRUE ;
    return FALSE ;
}

void del ( struct node *n )
{
    struct node *temp ;
    while ( n != NULL )
    {
        temp = n -> next ;
        free ( n ) ;
    }
n = temp;
}
}
Output:
/*enter the vertices: 9 
Enter the arry elments
0 1 -1 -1 -1 -1 -1 -1
1 0 1 -1 -1 -1 -1 -1
-1 1 0 1 -1 -1 -1 -1
-1 -1 1 0 1 1 -1 -1
-1 -1 -1 1 1 1 -1 -1
-1 -1 -1 -1 0 -1 1 -1
-1 -1 -1 -1 1 0 -1 1
-1 -1 -1 -1 1 -1 1 0
-1 -1 -1 -1 -1 1 1 0
4 node has maximum degree
No of edges connected are 3
diameter of given cycle c6 is 3
path the graph follows is shown below
1 2 3 4 5 6 7 8

4.3 Conclusion:

By using a BFS, here we are finding a Star $S_n$ consisting all the vertices of the
given graph and the Path $P_n$ consisting all the vertices of the given graph. Only few
classes of graphs have been tested here. We can consider some more class of graphs
for further analysis like House graph, Bull Graph.
Reference:


