BFSPath in C++

```
#include <iostream>
#include <vector>
#include <deque>

using namespace std;

// Edge structure representing an edge between two vertices
struct Edge {
    int src;
    int nbr;

    Edge(int src, int nbr) {
        this->src = src;
        this->nbr = nbr;
    }
};

// Pair structure to store vertex and path so far
struct Pair {
    int v;
    string psf;

    Pair(int v, string psf) : v(v), psf(psf) {}
};

// Function to add an edge between two vertices
void addEdge(vector<Edge>* graph, int v1, int v2) {
    graph[v1].push_back(Edge(v1, v2));
    graph[v2].push_back(Edge(v2, v1));
}

int main() {
    int vtces = 7; // Number of vertices
    vector<Edge>* graph = new vector<Edge>[vtces]; // Adjacency list of edges

    // Adding edges to the graph
    addEdge(graph, 0, 1);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    addEdge(graph, 0, 3);
    addEdge(graph, 3, 4);
    addEdge(graph, 4, 5);
    addEdge(graph, 5, 6);
    addEdge(graph, 4, 6);

    int src = 0; // Source vertex for BFS

    deque<Pair> q; // Queue for BFS
    vector<bool> visited(vtces, false); // Array to mark visited vertices

    q.push_back(Pair(src, to_string(src))); // Pushing source vertex with path so far
```

Graph Structure:

Edges (undirected):

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

This gives us the following adjacency list:

Vertex	Neighbors
0	1, 3
1	0, 2
2	1, 3
3	2, 0, 4
4	3, 5, 6
5	4, 6
6	5, 4

🧠 **BFS Behavior:**

- Queue type: deque
- Visited is marked **only when popped** (standard BFS behavior)
- Pair stores (vertex, path-so-far)
- Queue allows tracking of the shortest path from source

📜 **Dry Run Table:**

Step	Queue (Front → Back)	Visited	Output
1	(0, "0")	{}	
2	—	{0}	0 0
	Enqueue: (1, "01"), (3, "03")		
3	(1, "01"), (3, "03")	{0}	
4	—	{0, 1}	1 01
	Enqueue: (0, "010"), (2, "012")		
5	(3, "03"), (0, "010"), (2, "012")	{0, 1}	
6	—	{0, 1, 3}	3 03
	Enqueue: (2, "032"), (0, "030"), (4, "034")		
7	(0, "010"), (2, "012"), (2, "032"), (0, "030"), (4, "034")	{0, 1, 3}	
8	— 0 already visited → skip	{0, 1, 3}	
9	—	{0, 1, 2, 3}	2 012
	Enqueue: (1, "0121"), (3, "0123")		
10	— 2 already visited → skip		
11	— 0 already visited → skip		

```
    while (!q.empty()) {
        Pair rem = q.front();
        q.pop_front();

        if (visited[rem.v]) {
            continue;
        }
        visited[rem.v] = true;

        cout << rem.v << " " << rem.psf << endl; // Printing vertex and path so far

        // Iterating through all adjacent vertices
        for (Edge e : graph[rem.v]) {
            q.push_back(Pair(e.nbr, rem.psf + to_string(e.nbr))); // Adding adjacent vertices to queue
        }
    }

    delete[] graph; // Freeing dynamically allocated memory for graph
    return 0;
}
```

Step	Queue (Front → Back)	Visited	Output
12	—	{0,1,2,3,4}	4 034
	Enqueue: (3, "0343"), (5, "0345"), (6, "0346")		
13	— 1 already visited → skip		
14	— 3 already visited → skip		
15	—	{..., 5}	5 0345
	Enqueue: (4, "03454"), (6, "03456")		
16	—	{..., 6}	6 0346
	Enqueue: (5, "03465"), (4, "03464")		
...	All remaining vertices already visited → skip		

✅ **Final Output:**

(printed in order of **first encounter** in BFS)

0 0
1 01
3 03
2 012
4 034
5 0345
6 0346

Output:-
0 0
1 01
3 03
2 012
4 034
5 0345
6 0346

Hamiltonian Path and Cycle in C++

```
#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

// Structure to represent an edge in the
graph
struct Edge {
    int src;
    int nbr;
    int wt;

    Edge(int src, int nbr, int wt) {
        this->src = src;
        this->nbr = nbr;
        this->wt = wt;
    }
};

// Function to add an edge to the graph
void addEdge(vector<Edge>* graph, int
src, int nbr, int wt) {
    graph[src].push_back(Edge(src, nbr,
wt));
    graph[nbr].push_back(Edge(nbr, src,
wt)); // Assuming undirected graph
}

// Function to perform Hamiltonian path
and cycle calculation
void h(vector<Edge>* graph, int src,
unordered_set<int>& visited, string psf,
int originalSrc) {
    if (visited.size() == graph->size() - 1) {
        cout << psf;

        bool containsCycle = false;
        for (Edge& e : graph[src]) {
            if (e.nbr == originalSrc) {
                containsCycle = true;
                break;
            }
        }

        if (containsCycle) {
            cout << "*" << endl;
        } else {
            cout << "." << endl;
        }

        return;
    }

    visited.insert(src);
    for (Edge& e : graph[src]) {
        if (visited.find(e.nbr) ==
visited.end()) {
            h(graph, e.nbr, visited, psf +
to_string(e.nbr), originalSrc);
        }
```

Goal:

Explore all **Hamiltonian paths/cycles** starting from node 0.

🧩 Graph Summary:

Node	Neighbors
0	1, 3
1	0, 2
2	1, 3, 4
3	0, 2, 4
4	3, 5, 2
5	4

✅ Table Format:

Step	Current Node	Visited Set	Path So Far (psf)	Action
1	0	{0}	"0"	Start
2	1	{0,1}	"01"	0 → 1
3	2	{0,1,2}	"012"	1 → 2
4	3	{0,1,2,3}	"0123"	2 → 3
5	4	{0,1,2,3,4}	"01234"	3 → 4
6	5	{0,1,2,3,4,5}	"012345"	4 → 5
7	—	—	"012345."	6 vertices visited, no edge 5→0

➡ So we **print: 012345.**

Let's try another valid path:

Step	Current Node	Visited Set	Path So Far (psf)	Action
1	0	{0}	"0"	Start
2	3	{0,3}	"03"	0 → 3
3	2	{0,3,2}	"032"	3 → 2
4	1	{0,3,2,1}	"0321"	2 → 1
5	4	{0,3,2,1,4}	"03214"	2 → 4
6	5	{0,3,2,1,4,5}	"032145"	4 → 5
7	—	—	"032145."	No edge 5→0, just a path

➡ We print: 032145.

Let's do a cycle example:

Step	Current Node	Visited Set	Path So Far (psf)	Action
1	0	{0}	"0"	Start
2	3	{0,3}	"03"	0 → 3

```
    }
    visited.erase(src);
}

int main() {
    int vtces = 6; // Number of vertices
    //int edges = 7; // Number of edges

    // Create the graph using adjacency
list representation
    vector<Edge>* graph = new
vector<Edge>[vtces];

    // Add edges to the graph
    addEdge(graph, 0, 1, 10);
    addEdge(graph, 0, 3, 40);
    addEdge(graph, 1, 2, 10);
    addEdge(graph, 2, 3, 10);
    addEdge(graph, 3, 4, 2);
    addEdge(graph, 4, 5, 2);
    addEdge(graph, 2, 4, 3);

    int src = 0; // Source vertex

    // Perform Hamiltonian path and cycle
calculation
    unordered_set<int> visited;
    h(graph, src, visited, to_string(src),
src);

    delete[] graph; // Deallocate memory

    return 0;
}
```

Step	Current Node	Visited Set	Path So Far (psf)	Action
3	4	{0,3,4}	"034"	3 → 4
4	2	{0,3,4,2}	"0342"	4 → 2
5	1	{0,3,4,2,1}	"03421"	2 → 1
6	5	{0,3,4,2,1,5}	"034215"	4 → 5
7	—	—	"034215*"	Edge exists 5→0 → CYCLE ✓

➡ We print: 034215*

🔚 Summary of Dry Run:

Path	Hamiltonian	Cycle?
012345	✓	✗
032145	✓	✗
034215	✓	✓

Output:-
01*
03*

Print All Paths in C++

```
#include <iostream>
#include <vector>

using namespace std;

// Define the Edge structure
struct Edge {
    int src;
    int nbr;
    int wt;

    Edge(int s, int n, int w) {
        src = s;
        nbr = n;
        wt = w;
    }
};

// Function prototypes
void addEdge(vector<Edge>* graph, int
src, int nbr, int wt);
void printAllPaths(vector<Edge>* graph,
int src, int dest, vector<bool>& visited,
string psf);

int main() {
    int vtces = 6; // Number of vertices
    //int edges = 7; // Number of edges

    // Create the graph using vector of
vectors
    vector<Edge>* graph = new
vector<Edge>[vtces];

    // Add edges statically
    addEdge(graph, 0, 1, 10);
    addEdge(graph, 0, 3, 40);
    addEdge(graph, 1, 2, 10);
    addEdge(graph, 2, 3, 10);
    addEdge(graph, 3, 4, 2);
    addEdge(graph, 4, 5, 2);
    addEdge(graph, 2, 4, 3);

    int src = 0;   // Source vertex
    int dest = 5;  // Destination vertex

    // Array to track visited vertices
    vector<bool> visited(vtces, false);

    // Call the function to print all paths
from src to dest
    printAllPaths(graph, src, dest, visited,
to_string(src));

    return 0;
}

// Function to add an edge to the graph
void addEdge(vector<Edge>* graph, int
src, int nbr, int wt) {
    graph[src].emplace_back(src, nbr, wt);
```

Graph Structure:

Edges:

0 -- 1 (10)
0 -- 3 (40)
1 -- 2 (10)
2 -- 3 (10)
3 -- 4 (2)
4 -- 5 (2)
2 -- 4 (3)

This gives us the adjacency list:

Vertex	Neighbors
0	1, 3
1	0, 2
2	1, 3, 4
3	0, 2, 4
4	3, 5, 2
5	4

🎯 Goal:

Find all paths from src = 0 to dest = 5.

📜 Dry Run Table:

Recursive Call	Current src	Path So Far (psf)	Action
1	0	"0"	Explore neighbors 1, 3
2	1	"01"	Explore neighbors 2
3	2	"012"	Explore 3, 4
4	3	"0123"	Explore 4
5	4	"01234"	Explore 5
6	5	"012345"	✅ **Print this path**
Backtrack to 4			
Backtrack to 3			

```
    graph[nbr].emplace_back(nbr, src, wt);
}

// Function to print all paths from src to
dest
void printAllPaths(vector<Edge>* graph,
int src, int dest, vector<bool>& visited,
string psf) {
    if (src == dest) {
        cout << psf << endl;
        return;
    }

    visited[src] = true;
    for (Edge edge : graph[src]) {
        if (!visited[edge.nbr]) {
            printAllPaths(graph, edge.nbr,
dest, visited, psf + to_string(edge.nbr));
        }
    }
    visited[src] = false;
}
```

Recursive Call	Current src	Path So Far (psf)	Action
4 (alt)	4	"0124"	Explore 5
5	5	"01245"	✅ **Print this path**
Backtrack to 2			
Backtrack to 1			
Backtrack to 0			
2	3	"03"	Explore 2, 4
3	2	"032"	Explore 4
4	4	"0324"	Explore 5
5	5	"03245"	✅ **Print this path**
Backtrack to 3			
3 (alt)	4	"034"	Explore 5
4	5	"0345"	✅ **Print this path**

✅ **Final Output:**

012345
01245
03245
0345

Output:-

012345
01245
03245
0345