Shortest Path in a weighted Graph where weight of an edge is 1 or 2

Given a directed graph where every edge has weight as either 1 or 2, find the shortest path from a given source vertex 's' to a given destination vertex 't'. Expected time complexity is O(V+E).
A **Simple Solution** is to use Dijkstra's shortest path algorithm, we can get a shortest path in O(E + VLogV) time.

How to do it in O(V+E) time? The idea is to use BFS. One important observation about BFS is, the path used in BFS always has least number of edges between any two vertices. So if all edges are of same weight, we can use BFS to find the shortest path. For this problem, we can modify the graph and split all edges of weight 2 into two edges of weight 1 each. In the modified graph, we can use BFS to find the shortest path.

How many new intermediate vertices are needed? We need to add a new intermediate vertex for every source vertex. The reason is simple, if we add a intermediate vertex x between u and v and if we add same vertex between y and z, then new paths u to z and y to v are added to graph which might have note been there in original graph. Therefore in a graph with V vertices, we need V extra vertices.

Below is C++ implementation of above idea. In the below implementation 2*V vertices are created in a graph and for every edge (u, v), we split it into two edges (u, u+V) and (u+V, w). This way we make sure that a different intermediate vertex is added for every source vertex.

```
// Program to shortest path from a given source vertex 's' to
// a given destination vertex 't'.  Expected time complexity
// is O(V+E).
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // adjacency lists
public:
    Graph(int V);  // Constructor
    void addEdge(int v, int w, int weight); // adds an edge

    // finds shortest path from source vertex 's' to
    // destination vertex 'd'.
    int findShortestPath(int s, int d);

    // print shortest path from a source vertex 's' to
    // destination vertex 'd'.
    int printShortestPath(int parent[], int s, int d);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[2*V];
}

void Graph::addEdge(int v, int w, int weight)
{
    // split all edges of weight 2 into two
    // edges of weight 1 each.  The intermediate
    // vertex number is maximum vertex number + 1,
    // that is V.
    if (weight==2)
    {
        adj[v].push_back(v+V);
        adj[v+V].push_back(w);
    }
    else // Weight is 1
        adj[v].push_back(w); // Add w to v's list.
}
```

```
// To print the shortest path stored in parent[]
int Graph::printShortestPath(int parent[], int s, int d)
{
    static int level = 0;

    // If we reached root of shortest path tree
    if (parent[s] == -1)
    {
        cout << "Shortest Path between " << s << " and "
             << d << " is "  << s << " ";
        return level;
    }

    printShortestPath(parent, parent[s], d);

    level++;
    if (s < V)
        cout << s << " ";

    return level;
}

// This function mainly does BFS and prints the
// shortest path from src to dest. It is assumed
// that weight of every edge is 1
int Graph::findShortestPath(int src, int dest)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[2*V];
    int *parent = new int[2*V];

    // Initialize parent[] and visited[]
    for (int i = 0; i < 2*V; i++)
    {
        visited[i] = false;
        parent[i] = -1;
    }

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[src] = true;
    queue.push_back(src);

    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        int s = queue.front();

        if (s == dest)
            return printShortestPath(parent, s, dest);

        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it
        // visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
                parent[*i] = s;
            }
        }
    }
}
```

```
// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    int V = 4;
    Graph g(V);
    g.addEdge(0, 1, 2);
    g.addEdge(0, 2, 2);
    g.addEdge(1, 2, 1);
    g.addEdge(1, 3, 1);
    g.addEdge(2, 0, 1);
    g.addEdge(2, 3, 2);
    g.addEdge(3, 3, 2);

    int src = 0, dest = 3;
    cout << "\nShortest Distance between " << src
         << " and " << dest << " is "
         << g.findShortestPath(src, dest);

    return 0;
}
```

Output :

```
Shortest Path between 0 and 3 is 0 1 3
Shortest Distance between 0 and 3 is 3
```

How is this approach O(V+E)? In worst case, all edges are of weight 2 and we need to do O(E) operations to split all edges and 2V vertices, so the time complexity becomes O(E) + O(V+E) which is O(V+E).