Pseudocode of Find_Path(v1,v2)

Problem: You are given an undirected graph G (V, E), where 'V' is the set of vertices and 'E' is the set of edges present in the graph. n=|V|=Number of vertices. Assume vertices are numbered from 0 to n-1. 0 is the source and (n-1) is the sink vertex.

Input: Two integers 'v1' (source node) and 'v2'(sink node) denoting vertices of the graph

Output: If there exists a Path from 'v1' to 'v2' then update the Graph G accordingly and return the Bottleneck Capacity value (which can be used to increase the flow.

Return 0, otherwise i.e. if there is no Augmenting Path from 'v1' to 'v2'

Description: In this approach, iterate through the whole graph using BFS and whenever you encounter a new node with positive edge weight, update the parent of the new node by the current node. Enqueue and Dequeue functions are defined as shown below.
Assume WHITE=0, GRAY=1, BLACK=2, head, tail are global variables. Color, q and the parent are the global arrays of size equals to the Number of Nodes in the Graph, i.e. n. Enqueue() colors the vertex as Gray and Dequeue() makes the vertex Black. If there is an augmenting path from 'v1' to 'v2' then 'v2' will be enqueued and dequeued eventually and hence 'v2' will be colored black. If there is an augmenting path, the algorithm returns Bottleneck capacity and returns it after updating the Graph G, otherwise it returns 0.

Enqueue	Dequeue

void Enqueue (int x) {
  q[tail] = x;
  tail++;
  color[x] = **GRAY**;
}
	int Dequeue () {
	  int x = q[head];
	  head++;
	  color[x] = **BLACK**;
	  return x;
	}

Find_Path(v1,v2):

1. Initialize the color of each vertex as WHITE.
2. Head=0, Tail=0
3. Initialize a list **Parent** with -1 to store the parent of nodes.
4. Create a queue for storing the nodes to iterate through breadth-first search.
5. Enqueue **v1** to the queue. i.e. Enqueue(v1)
6. While (Head!=Tail)

```
{
    u=Dequeue()
     /* Search all adjacent white nodes v. If the capacity from u to v in the
         residual network is positive enqueue v.*/
    for every node v, which is adjacent to u
    {
     If (v is not visited yet and (u, v) edge is having positive weight in G)
        {
            Enqueue v in the Queue
            Parent[v]=u
        }
    }
}
```

7. /*If the color of the target node v2 is black now, it means that we reached it.*/

```
  If ( color[v2]==BLACK)
  {    // Determine the amount by which we can increment the flow.
       increment=INT_MAX;
       // Traversal of Augmenting Path to compute the Bottleneck
       for (u=n-1; Parent[u]>=0; u=Parent[u])          {
        increment =min(increment,G[Parent[u]][u]);
       }
    // Now Update G
       for (u=n-1; Parent[u]>=0; u=pred[u])
      {
          //Subtract from Forward Edges of Path
           G[Parent[u]][u] -= increment;
        //Add to Backward Edges of the Path
           G[u][Parent[u]] += increment;  //BackEdges
       }

     return increment; //Bottleneck capacity

  }
  Else
     return 0; //No Augmenting Path
```