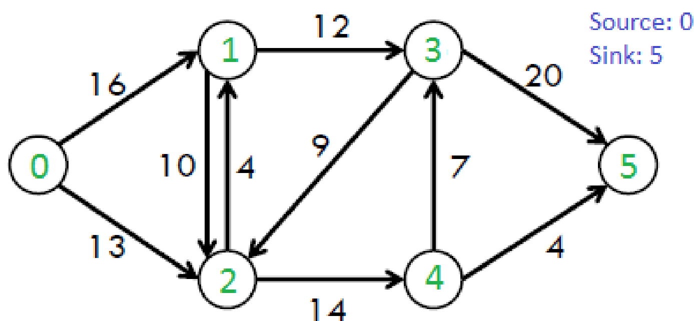
Ford-Fulkerson Algorithm for Maximum Flow Problem

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source* ‘s’ and *sink* ‘t’ in the graph, find the maximum possible flow from s to t with following constraints:

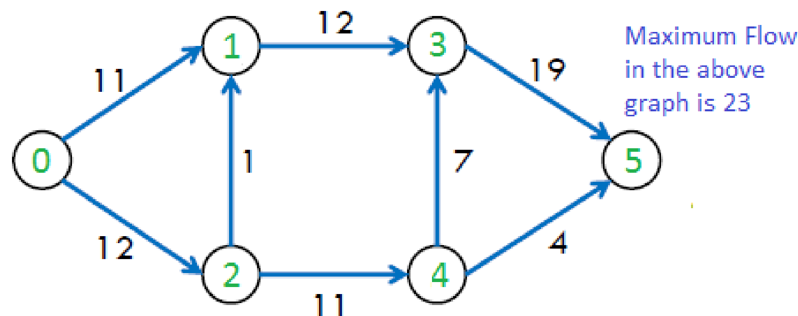
a) Flow on an edge doesn’t exceed the given capacity of the edge.

b) Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.




```
Ford-Fulkerson Algorithm
The following is simple idea of Ford-Fulkerson algorithm:
1) Start with initial flow as 0.
2) While there is a augmenting path from source to sink.
           Add this path-flow to flow.
3) Return flow.
```

Time Complexity: Time complexity of the above algorithm is O(max_flow * E). We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max_flow * E).

How to implement the above simple algorithm?

Let us first define the concept of Residual Graph which is needed for understanding the implementation.

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called ***residual capacity*** which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge.
Let us now talk about implementation details. Residual capacity is 0 if there is no edge between to vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation. Using BFS, we can find out if there is a path from source to sink. BFS also builds parent[] array. Using the parent[] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.
The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges We need to add path flow along reverse edges because may later need to send flow in reverse direction (See following video for example).

http://www.youtube.com/watch?v=-8MwfgB-lyM

Following is C++ implementation of Ford-Fulkerson algorithm. To keep things simple, graph is represented as a 2D matrix.

```
// C++ program for implementation of Ford Fulkerson a
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to
  residual graph. Also fills parent[] to store the pa
bool bfs(int rGraph[V][V], int s, int t, int parent[]
{
    // Create a visited array and mark all vertices a
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
```

```
    {
        int u = q.front();
        q.pop();

        for (int v=0; v<V; v++)
        {
            if (visited[v]==false && rGraph[u][v] > 0
            {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    // If we reached sink in BFS starting from source
    // true, else false
    return (visited[t] == true);
}

// Returns tne maximum flow from s to t in the given
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual
    // given capacities in the original graph as resi
    // in residual graph
    int rGraph[V][V]; // Residual graph where rGraph[
                      // residual capacity of edge fro
                      // is an edge. If rGraph[i][j] i
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
             rGraph[u][v] = graph[u][v];

    int parent[V];  // This array is filled by BFS an

    int max_flow = 0;  // There is no flow initially

    // Augment the flow while tere is path from sourc
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edhe
        // path filled by BFS. Or we can say find the
        // through the path found.
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }
```

```
        // update residual capacities of the edges and
        // along the path
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow
    return max_flow;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above examp
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0}
                      };

    cout << "The maximum possible flow is " << fordFu

    return 0;
}
```

Output:

```
The maximum possible flow is 23
```

The above implementation of Ford Fulkerson Algorithm is called **Edmonds-Karp Algorithm**. The idea of Edmonds-Karp is to use BFS in Ford Fulkerson implementation as BFS always picks a path with minimum number of edges. When BFS is used, the worst case time complexity can be reduced to $O(VE^2)$. The above implementation uses adjacency matrix representation though where BFS takes $O(V^2)$ time, the time complexity of the above implementation is $O(EV^3)$ (Refer CLRS book for proof of time complexity)

This is an important problem as it arises in many practical situations. Examples include, maximizing the transportation with given traffic limits, maximizing packet flow in computer networks.

Exercise: