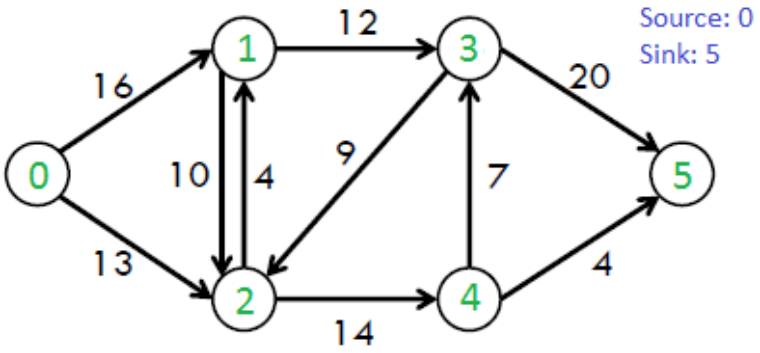
Graphs
Lesson 9

1. Max Flow Problem Introduction
2. Ford-Fulkerson Algorithm for Maximum Flow Problem
3. Find minimum s-t cut in a flow network
4. Dinic’s algorithm for Maximum Flow
5. Eulerian path and circuit

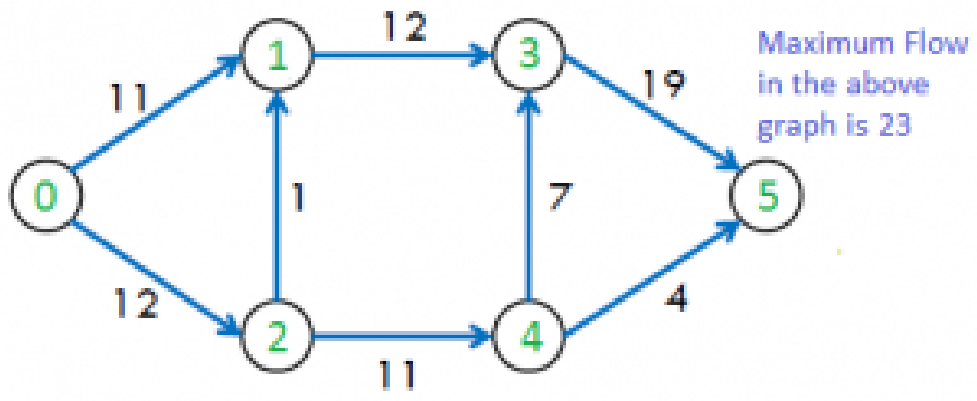
Max Flow Problem Introduction

Maximum flow problems involve finding a feasible flow through a single-source, single-sink flow network that is maximum.

Let's take an image to explain what the above definition wants to say.



Each edge is labeled with capacity, the maximum amount of stuff that it can carry. The goal is to figure out how much stuff can be pushed from the vertex s(source) to the vertex t(sink).



.

maximum flow possible is : **23**

Following are different approaches to solve the problem :

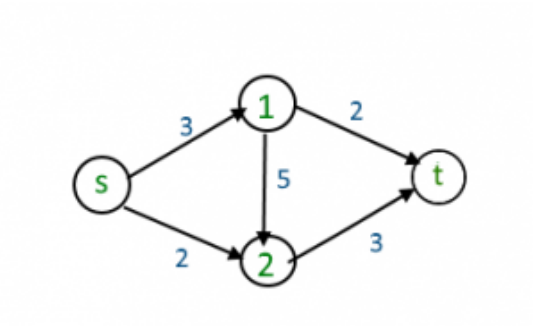
1. Naive Greedy Algorithm Approach (May not produce an optimal or correct result)

Greedy approach to the maximum flow problem is to start with the all-zero flow and greedily produce flows with ever-higher value. The natural way to proceed from one to the next is to send more flow on some path from s to t

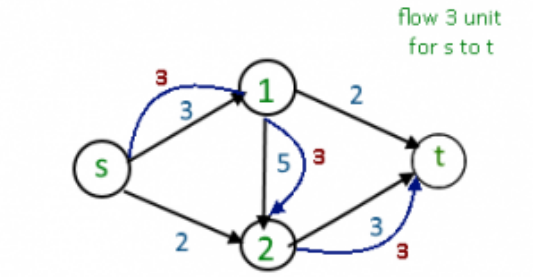
How does the Greedy approach work to find the maximum flow :

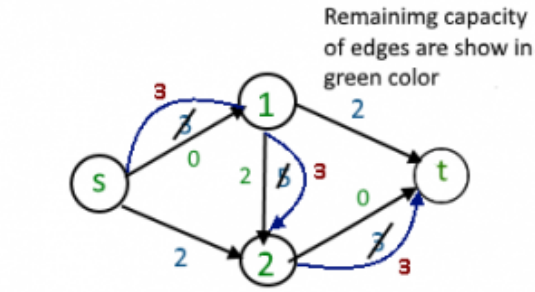
```
E number of edge
f(e) flow of edge
C(e) capacity of edge

1) Initialize : max_flow = 0
        f(e) = 0 for every edge 'e' in E

2) Repeat search for an s-t path P while it exists.
  a) Find if there is a path from s to t using BFS
    or DFS. A path exists if f(e) < C(e) for
    every edge e on the path.
  b) If no path found, return max_flow.
  c) Else find minimum edge value for path P

    // Our flow is limited by least remaining
    // capacity edge on path P.
    (i) flow = min(C(e)- f(e)) for path P ]
        max_flow += flow
    (ii) For all edge e of path increment flow
        f(e) += flow

3) Return max_flow
```

Note that the path search just needs to determine whether or not there is an s-t path in the subgraph of edges e with f(e) < C(e). This is easily done in linear time using BFS or DFS.

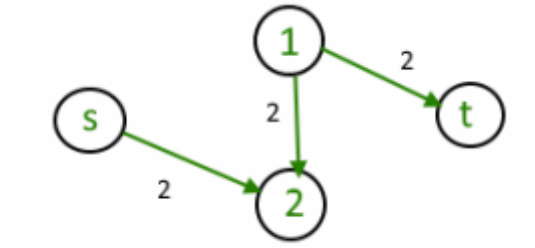


There is a path from source (s) to sink(t) [s -> 1 -> 2 -> t] with maximum flow 3 units (path shown in blue color)





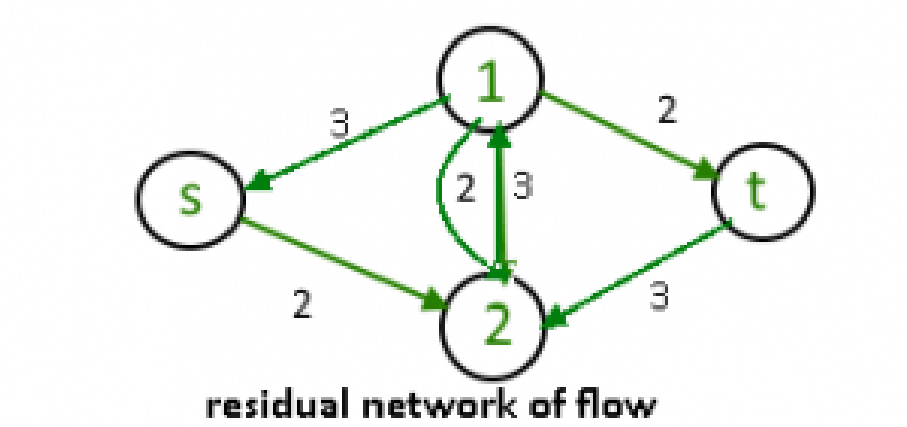
After removing all useless edges from the graph it looks like



For above graph there is no path from source to sink so maximum flow : 3 unit But maximum flow is 5 unit. to overcome from this issue we use residual Graph.

2. Residual Graphs

The idea is to extend the naive greedy algorithm by allowing “undo” operations. For example, from the point where this algorithm gets stuck in above image, we’d like to route two more units of flow along the edge (s, 2), then backward along the edge (1, 2), undoing 2 of the 3 units we routed the previous iteration, and finally along the edge (1,t)



residual network of flow

backward edge : (f(e)) and forward edge : (C(e) – f(e))

We need a way of formally specifying the allowable “undo” operations. This motivates the following simple but important definition of a residual network. The idea is that, given a graph G and a flow f in it, we form a new flow network G_f that has the same vertex set of G and that has two edges for each edge of G. An edge e = (1,2) of G that carries flow f(e) and has capacity C(e) (for above image) spawns a “forward edge” of G_f with capacity C(e)-f(e) (the room remaining) and a “backward edge” (2,1) of G_f with capacity f(e) (the amount of previously routed flow that can be undone). source(s)- sink(t) paths with f(e) < C(e) for all edges, as searched for by the naive greedy algorithm, corresponding to the special case of s-t paths of G_f that comprise only forward edges.

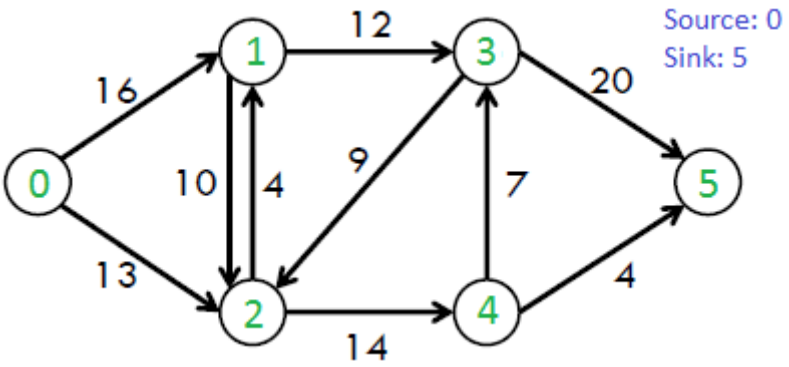
Ford-Fulkerson Algorithm for Maximum Flow Problem

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

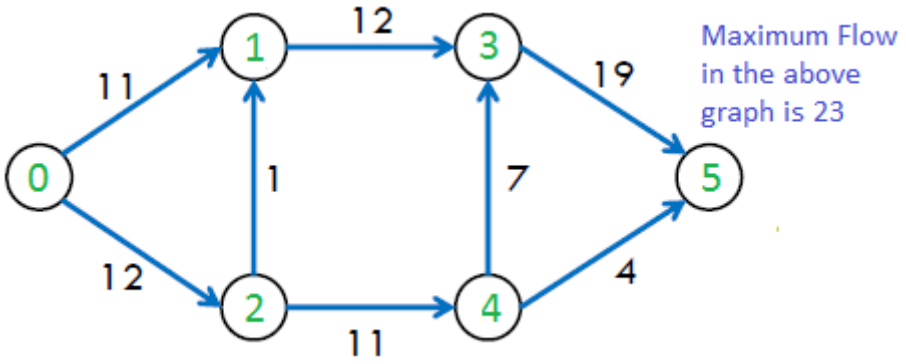
a) Flow on an edge doesn't exceed the given capacity of the edge.

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

For example, consider the following graph from the CLRS book.



The maximum possible flow in the above graph is 23.



Ford-Fulkerson Algorithm

The following is a simple idea of the Ford-Fulkerson algorithm:

1) Start with initial flow as 0.

2) While there is an 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 the 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 two 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 link for example).

https://www.geeksforgeeks.org/max-flow-problem-introduction/

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

```
1.   bool BFS(vector<vector<int>> adj, int N, int parent[])
2.   {
3.     bool visited[N+1];
4.     memset(visited,false,sizeof(visited));
5.     queue<int> q;
6.     q.push(1);
7.     visited[1]=true;
8.     while(!q.empty())
9.     {
10.       int u=q.front();
11.       q.pop();
12.       for(int i=1; i<=N; i++)
13.       {
14.          if(!visited[i] && adj[u][i]>0)
15.          {
16.             parent[i]=u;
17.             if(i==N)
18.                return true;
19.             visited[i]=true;
20.             q.push(i);
21.          }
22.       }
```

```
    }
    return false;
}
int bottle_neck_capacity(vector<vector<int>> adj, int N, int parent[])
{
    int flow=INT_MAX;
    for(int i=N; parent[i]!=-1; i=parent[i])
    {
        flow=min(flow,adj[parent[i]][i]);
    }
    return flow;
}
class Solution
{
public:
    int solve(int N,int M,vector<vector<int>> Edges)
    {
        // code here
        vector<vector<int>> adj(N+1,vector<int>(N+1,0));
        int i;
        for(i=0; i<M; i++)
        {
            adj[Edges[i][0]][Edges[i][1]]+=Edges[i][2];
            adj[Edges[i][1]][Edges[i][0]]+=Edges[i][2];
        }
        int parent[N+1];
        parent[1]=-1;
        int max_flow=0;
        while(BFS(adj,N,parent))
        {
            int min_flow=bottle_neck_capacity(adj,N,parent);
            max_flow+=min_flow;
            for(i=N; parent[i]!=-1; i=parent[i])
            {
```

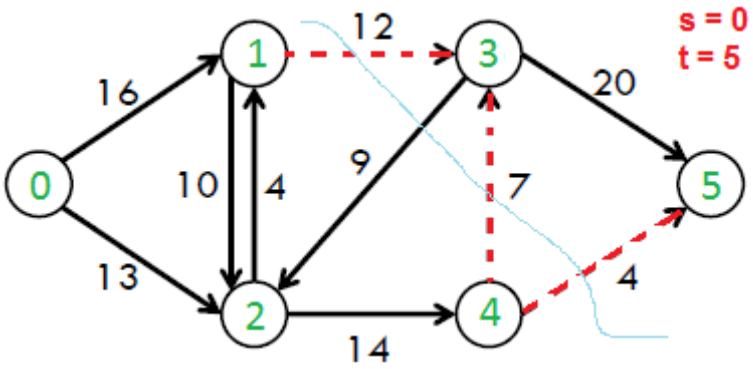
```
57.                 adj[parent[i]][i]-=min_flow;
58.                 adj[i][parent[i]]+=max_flow;
59.             }
60.         }
61.         return max_flow;
62.     }
63. };
```

Find minimum s-t cut in a flow network

In a flow network, an s-t cut is a cut that requires the source ‘s’ and the sink ‘t’ to be in different subsets, and it consists of edges going from the source’s side to the sink’s side. The capacity of an s-t cut is defined by the sum of the capacity of each edge in the cut-set. (Source: Wiki)

The problem discussed here is to find minimum capacity s-t cut of the given network. Expected output is all edges of the minimum cut.

For example, in the following flow network, example s-t cuts are {{0 ,1}, {0, 2}}, {{0, 2}, {1, 2}, {1, 3}}, etc. The minimum s-t cut is {{1, 3}, {4, 3}, {4 5}} which has capacity as 12+7+4 = 23.



Minimum Cut and Maximum Flow

Like Maximum Bipartite Matching, this is another problem which can solved using Ford-Fulkerson Algorithm. This is based on max-flow min-cut theorem.

The max-flow min-cut theorem states that in a flow network, the amount of maximum flow is equal to capacity of the minimum cut. See CLRS book for proof of this theorem.

From Ford-Fulkerson, we get capacity of minimum cut. How to print all edges that form the minimum cut? The idea is to use residual graph.

Following are steps to print all edges of the minimum cut.

1) Run Ford-Fulkerson algorithm and consider the final residual graph.

2) Find the set of vertices that are reachable from the source in the residual graph.

3) All edges which are from a reachable vertex to non-reachable vertex are minimum cut edges. Print all such edges.

Following is the implementation of the above approach.

```
1.  bool BFS(vector<vector<int>> &A, int S, int T, int N, int parent[])
2.  {
3.    bool visited[N];
4.    memset(visited,false,sizeof(visited));
5.    queue<int> q;
6.    q.push(S);
7.    visited[S]=true;
8.    while(!q.empty())
9.    {
10.     S=q.front();
11.     q.pop();
12.     for(int i=0; i<N; i++)
13.     {
14.       if(!visited[i] && A[S][i]>0)
15.       {
16.         parent[i]=S;
17.         if(i==T)
18.           return true;
19.         visited[i]=true;
20.         q.push(i);
21.       }
22.     }
23.   }
24.   return false;
25. }
26. int bottle_neck_capacity(vector<vector<int>> A, int T, int N, int parent[])
27. {
28.   int flow=INT_MAX;
29.   for(int i=T; parent[i]!=-1; i=parent[i])
30.     flow=min(flow,A[parent[i]][i]);
31.   return flow;
32. }
33. void dfs(vector<vector<int>> A, bool visited[], int N, int v)
34. {
35.   visited[v]=true;
36.   for(int i=0; i<N; i++)
37.   {
38.     if(!visited[i] && A[v][i]!=0)
39.       dfs(A,visited,N, i);
40.   }
```

```
}
vector<int> minimumCut(vector<vector<int>> &A, int S, int T, int N){
 vector<vector<int>> G(N,vector<int>(N,0));
  for(int i=0; i<N; i++)
  {
    for(int j=0; j<N; j++)
      G[i][j]=A[i][j];
  }
  int parent[N];
  parent[S]=-1;
  int max_flow=0;
  while(BFS(A,S,T,N,parent))
  {
    int min_flow=bottle_neck_capacity(A,T,N,parent);
    max_flow+=min_flow;
    for(int i=T; i!=S; i=parent[i])
    {
      A[parent[i]][i]-=min_flow;
      A[i][parent[i]]+=min_flow;
    }
  }
  bool visited[N];
  memset(visited,false,sizeof(visited));
  dfs(A,visited,N,S);
  vector<int> ans;
  for(int i=0; i<N; i++)
  {
    for(int j=0; j<N; j++)
    {
      if(visited[i] && !visited[j] && G[i][j]>0)
      {
        ans.push_back(i);
        ans.push_back(j);
      }
    }
  }
  if(ans.size()==0)
    ans.push_back(-1);
  return ans;
}
```

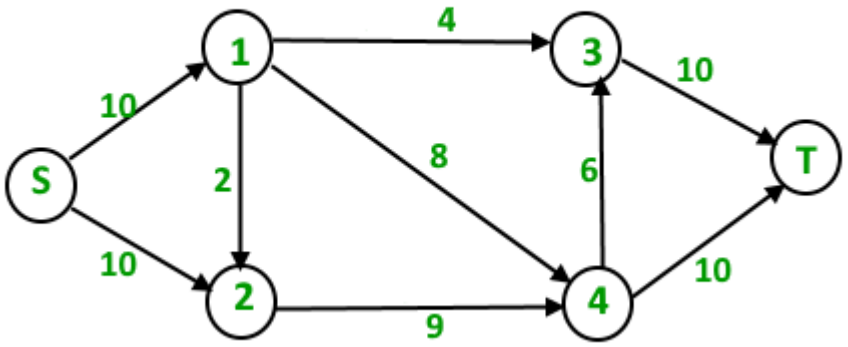
Dinic's algorithm for Maximum Flow

Problem Statement :

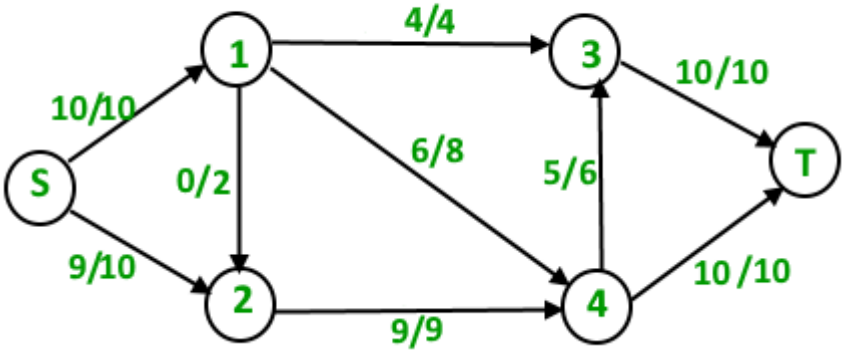
Given a graph which represents a flow network where every edge has a capacity. Given two vertices source 's' and sink 't' in the graph, find the maximum possible flow from s to t with following constraints :

1. Flow on an edge doesn't exceed the given capacity of the edge.
2. Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, in the following input graph,



the maximum s-t flow is 19 which is shown below.



Background :

1. Max Flow Problem Introduction : We introduced Maximum Flow problem, discussed Greedy Algorithm and introduced residual graph.
2. Ford-Fulkerson Algorithm and Edmond Karp Implementation : We discussed Ford-Fulkerson algorithm and its implementation. We also discussed residual graph in detail.

Time complexity of Edmond Karp Implementation is $O(VE^2)$. In this post, a new Dinic's algorithm is discussed which is a faster algorithm and takes $O(EV^2)$.

Like Edmond Karp's algorithm, Dinic's algorithm uses following concepts :

1. A flow is maximum if there is no **s** to **t** path in residual graph.
2. BFS is used in a loop. There is a difference though in the way we use BFS in both algorithms.

In Edmond's Karp algorithm, we use BFS to find an augmenting path and send flow across this path. In Dinic's algorithm, we use BFS to check if more flow is possible and to construct level graph. In **level graph**, we assign levels to all nodes, level of a node is shortest distance (in terms of number of edges) of the node from source. Once level graph is constructed, we send multiple flows using this level graph. This is the reason it works better than Edmond Karp. In Edmond Karp, we send only flow that is send across the path found by BFS.

Outline of Dinic's algorithm :

1) Initialize residual graph G as given graph.

1) Do BFS of G to construct a level graph (or assign levels to vertices) and also check if more flow is possible.

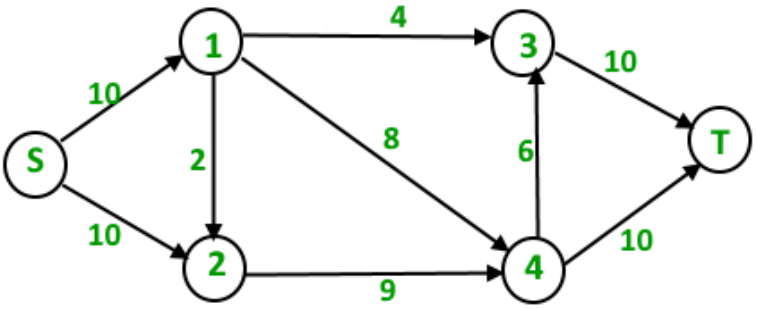
a) If more flow is not possible, then return.

b) Send multiple flows in G using level graph until blocking flow is reached. Here **using level graph** means, in every flow,levels of path nodes should be 0, 1, 2...(in order) from s to t.

A flow is **Blocking Flow** if no more flow can be sent using level graph, i.e., no more s-t path exists such that path vertices have current levels 0, 1, 2… in order. Blocking Flow can be seen same as maximum flow path in Greedy algorithm discussed here.

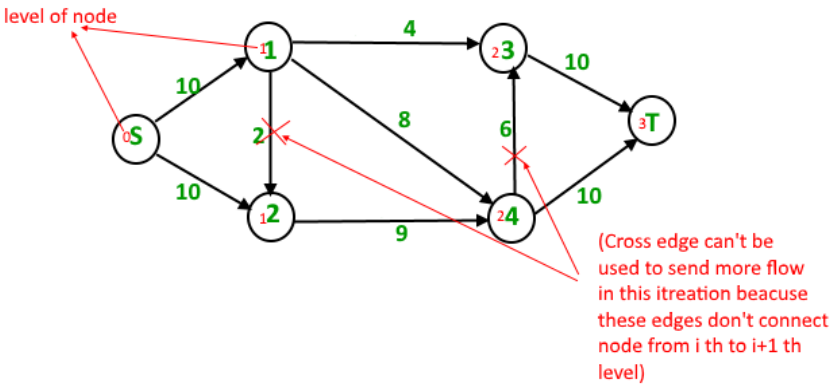
Illustration :

Initial Residual Graph (Same as given Graph)



Total Flow = 0

First Iteration : We assign levels to all nodes using BFS. We also check if more flow is possible (or there is a s-t path in residual graph).



Now we find blocking flow using levels (means every flow path should have levels as 0, 1, 2, 3). We send three flows together. This is where it is optimized compared to Edmond Karp where we send one flow at a time.

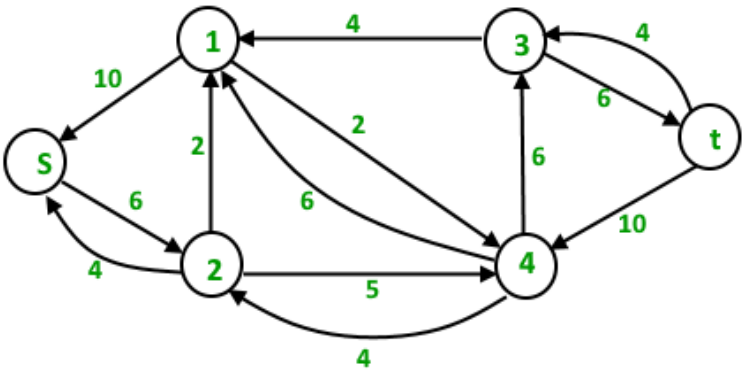
4 units of flow on path s – 1 – 3 – t.

6 units of flow on path s – 1 – 4 – t.

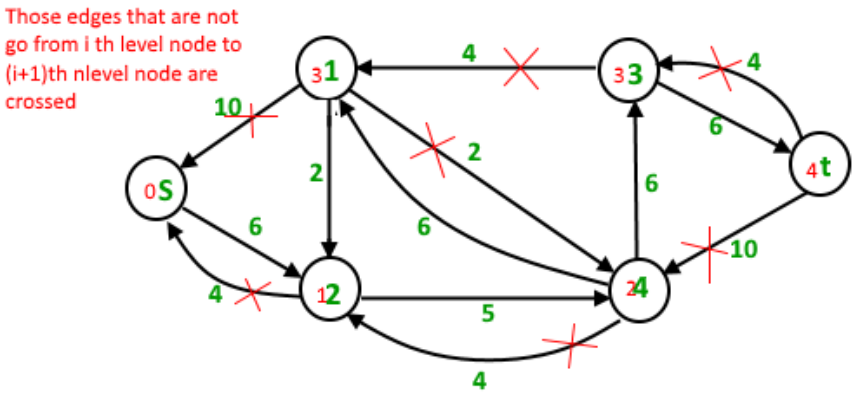
4 units of flow on path s – 2 – 4 – t.

Total flow = Total flow + 4 + 6 + 4 = 14

After one iteration, residual graph changes to following.



Second Iteration : We assign new levels to all nodes using BFS of above modified residual graph. We also check if more flow is possible (or there is a s-t path in residual graph).

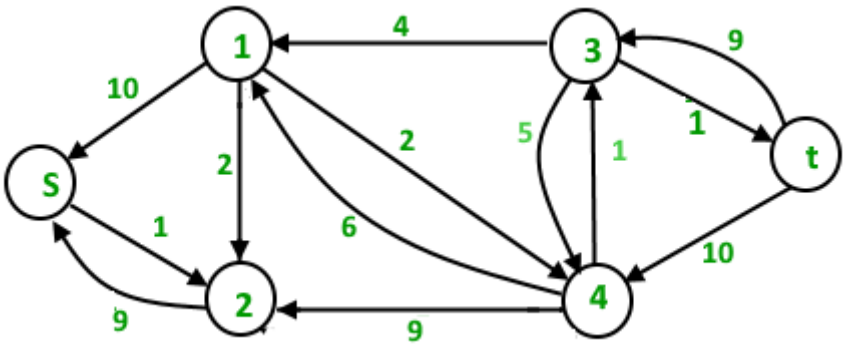


Now we find blocking flow using levels (means every flow path should have levels as 0, 1, 2, 3, 4). We can send only one flow this time.

5 units of flow on path s – 2 – 4 – 3 – t

Total flow = Total flow + 5 = 19

The new residual graph is



Third Iteration : We run BFS and create a level graph. We also check if more flow is possible and proceed only if possible. This time there is no s-t path in residual graph, so we terminate the algorithm.

Implementation :

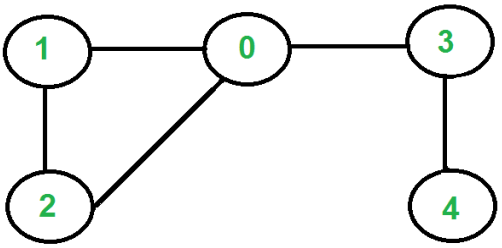
Below is c++ implementation of Dinic's algorithm:

```
1.  Bool BFS(int s, int t)
2.  {
3.    for (int i = 0 ; i < V ; i++)
4.       level[i] = -1;
5.    level[s] = 0;  // Level of source vertex
6.   list< int > q;
7.    q.push_back(s);
8.    vector<Edge>::iterator i ;
9.    while (!q.empty())
10.   {
11.      int u = q.front();
12.      q.pop_front();
13.      for (i = adj[u].begin(); i != adj[u].end(); i++)
14.      {
15.         Edge &e = *i;
16.         if (level[e.v] < 0  && e.flow < e.C)
17.         {level[e.v] = level[u] + 1;
18.            q.push_back(e.v);
19.         }
20.      }
21.   }
22.   return level[t] < 0 ? false : true ;
23. }
24. int sendFlow(int u, int flow, int t, int start[])
25. {
26.   if (u == t)
```

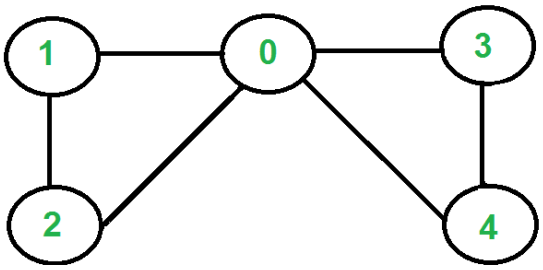
```
27.         return flow;
28.     for (  ; start[u] < adj[u].size(); start[u]++)
29.       {
30.         Edge &e = adj[u][start[u]];
31.         if (level[e.v] == level[u]+1 && e.flow < e.C)
32.         {
33.             // find minimum flow from u to t
34.             int curr_flow = min(flow, e.C - e.flow);
35.             int temp_flow = sendFlow(e.v, curr_flow, t, start);
36.                   if (temp_flow > 0)
37.             {
38.                 e.flow += temp_flow;
39.                     adj[e.v][e.rev].flow -= temp_flow;
40.                 return temp_flow;
41.             }
42.         }
43.       }
44.       return 0;
45.     }
46.     int DinicMaxflow(int s, int t)
47.     {
48.       if (s == t)
49.         return -1;
50.       int total = 0;  // Initialize result
51.       while (BFS(s, t) == true)
52.       {
53.         int *start = new int[V+1] {0};
54.         while (int flow = sendFlow(s, INT_MAX, t, start))
55.             total += flow;
56.       }
57.       return total;
58.     }
```

Eulerian path and circuit for undirected graph

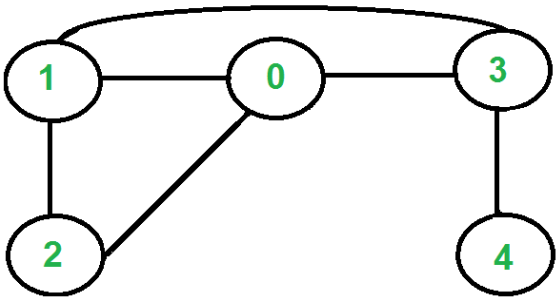
Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.



The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2" Note that all vertices have even degree



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

How to find whether a given graph is Eulerian or not?

The problem is same as following question. “Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once”.

A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to Hamiltonian Path which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in O(V+E) time.

Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

Eulerian Cycle

An undirected graph has Eulerian cycle if following two conditions are true.

….a) All vertices with non-zero degree are connected. We don’t care about vertices with zero degree because they don’t belong to Eulerian Cycle or Path (we only consider all edges).

….b) All vertices have even degree.

Eulerian Path

An undirected graph has Eulerian Path if following two conditions are true.

….a) Same as condition (a) for Eulerian Cycle

….b) If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note that a graph with no edges is considered Eulerian because there are no edges to traverse.

How does this work?

In Eulerian path, each time we visit a vertex v, we walk through two unvisited edges with one end point as v. Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

```
1.    bool Graph::isConnected()
2.    {
3.      // Mark all the vertices as not visited
4.      bool visited[V];
5.      int i;
6.      for (i = 0; i < V; i++)
7.          visited[i] = false;
8.
9.      // Find a vertex with non-zero degree
10.     for (i = 0; i < V; i++)
11.         if (adj[i].size() != 0)
12.             break;
13.
14.     // If there are no edges in the graph, return true
15.     if (i == V)
16.         return true;
17.
18.     // Start DFS traversal from a vertex with non-zero degree
19.     DFSUtil(i, visited);
20.
21.     // Check if all non-zero degree vertices are visited
22.     for (i = 0; i < V; i++)
23.       if (visited[i] == false && adj[i].size() > 0)
24.             return false;
25.
26.     return true;
```

```
27.    }
28.
29.    /* The function returns one of the following values
30.       0 --> If graph is not Eulerian
31.       1 --> If graph has an Euler path (Semi-Eulerian)
32.       2 --> If graph has an Euler Circuit (Eulerian)  */
33.    int Graph::isEulerian()
34.    {
35.       // Check if all non-zero degree vertices are connected
36.       if (isConnected() == false)
37.          return 0;
38.
39.       // Count vertices with odd degree
40.       int odd = 0;
41.       for (int i = 0; i < V; i++)
42.          if (adj[i].size() & 1)
43.             odd++;
44.
45.       // If count is more than 2, then graph is not Eulerian
46.       if (odd > 2)
47.          return 0;
48.
49.       // If odd count is 2, then semi-eulerian.
50.       // If odd count is 0, then eulerian
51.       // Note that odd count can never be 1 for undirected graph
52.       return (odd)? 1 : 2;
53.    }
54.
55.    // Function to run test cases
56.    void test(Graph &g)
57.    {
58.       int res = g.isEulerian();
59.       if (res == 0)
60.          cout << "graph is not Eulerian\n";
```

```
61.      else if (res == 1)
62.          cout << "graph has a Euler path\n";
63.      else
64.          cout << "graph has a Euler cycle\n";
65.  }
```