Push Relabel Algorithm | Set 2 (Implementation)

We strongly recommend to refer below article before moving on to this article.

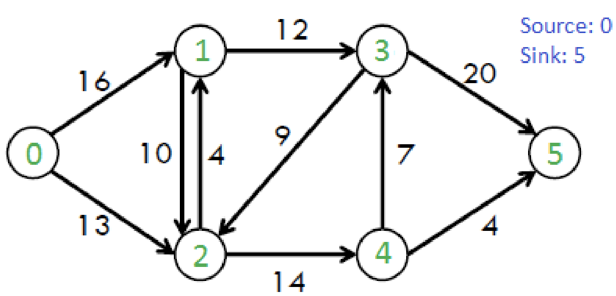
Push Relabel Algorithm | Set 1 (Introduction and Illustration)

Problem Statement : 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:

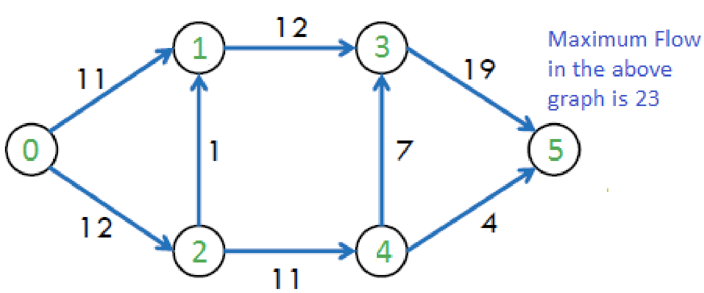
a) Flow on an edge doesn't exceed the given capacity of the edge.

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.




```
Push-Relabel Algorithm
1) Initialize PreFlow : Initialize Flows and Heights

2) While it is possible to perform a Push() or Relablel() on a vertex
   // Or while there is a vertex that has excess flow
           Do Push() or Relabel()

// At this point all vertices have Excess Flow as 0 (Except source
```

Below are main operations performed in Push Relabel algorithm.

There are three main operations in Push-Relabel Algorithm

1. **Initialize PreFlow()** It initializes heights and flows of all vertices.

   ```
   Preflow()
   1) Initialize height and flow of every vertex as 0.
   2) Initialize height of source vertex equal to total
      number of vertices in graph.
   3) Initialize flow of every edge as 0.
   4) For all vertices adjacent to source s, flow and
      excess flow is equal to capacity initially.
   ```

2. **Push()** is used to make the flow from a node which has excess flow. If a vertex has excess flow and there is an adjacent with smaller height (in residual graph), we push the flow from the vertex to the adjacent with lower height. The amount of pushed flow through the pipe (edge) is equal to the minimum of excess flow and capacity of edge.
3. **Relabel()** operation is used when a vertex has excess flow and none of its adjacent is at lower height. We basically increase height of the vertex so that we can perform push(). To increase height, we pick the minimum height adjacent (in residual graph, i.e., an adjacent to whom we can add flow) and add 1 to it.

Implementation:

The following implementation uses below structure for representing a flow network.

```
struct Vertex
{
   int h;      // Height of node
   int e_flow; // Excess Flow
}

struct Edge
{
   int u, v; // Edge is from u to v
   int flow; // Current flow
   int capacity;
}

class Graph
{
   Edge edge[];   // Array of edges
   Vertex ver[];  // Array of vertices
}
```

The below code uses given graph itself as a flow network and residual graph. We have not created a separate graph for residual graph and have used the same graph for simplicity.

```
// C++ program to implement push-relabel algorithm for
// getting maximum flow of graph
#include <bits/stdc++.h>
using namespace std;
```

```
    // An edge u--->v has start vertex as u and end
    // vertex as v.
    int u, v;

    Edge(int flow, int capacity, int u, int v)
    {
        this->flow = flow;
        this->capacity = capacity;
        this->u = u;
        this->v = v;
    }
};

// Represent a Vertex
struct Vertex
{
    int h, e_flow;

    Vertex(int h, int e_flow)
    {
        this->h = h;
        this->e_flow = e_flow;
    }
};

// To represent a flow network
class Graph
{
    int V;    // No. of vertices
    vector<Vertex> ver;
    vector<Edge> edge;

    // Function to push excess flow from u
    bool push(int u);

    // Function to relabel a vertex u
    void relabel(int u);

    // This function is called to initialize
    // preflow
    void preflow(int s);

    // Function to reverse edge
    void updateReverseEdgeFlow(int i, int flow);

public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // returns maximum flow from s to t
    int getMaxFlow(int s, int t);
};

Graph::Graph(int V)
{
    this->V = V;

    // all vertices are initialized with 0 height
    // and 0 excess flow
    for (int i = 0; i < V; i++)
        ver.push_back(Vertex(0, 0));
}
```

```
    edge.push_back(Edge(0, capacity, u, v));
}

void Graph::preflow(int s)
{
    // Making h of source Vertex equal to no. of vertices
    // Height of other vertices is 0.
    ver[s].h = ver.size();

    //
    for (int i = 0; i < edge.size(); i++)
    {
        // If current edge goes from source
        if (edge[i].u == s)
        {
            // Flow is equal to capacity
            edge[i].flow = edge[i].capacity;

            // Initialize excess flow for adjacent v
            ver[edge[i].v].e_flow += edge[i].flow;

            // Add an edge from v to s in residual graph with
            // capacity equal to 0
            edge.push_back(Edge(-edge[i].flow, 0, edge[i].v, s));
        }
    }
}

// returns index of overflowing Vertex
int overFlowVertex(vector<Vertex>& ver)
{
    for (int i = 1; i < ver.size() - 1; i++)
       if (ver[i].e_flow > 0)
            return i;

    // -1 if no overflowing Vertex
    return -1;
}

// Update reverse flow for flow added on ith Edge
void Graph::updateReverseEdgeFlow(int i, int flow)
{
    int u = edge[i].v, v = edge[i].u;

    for (int j = 0; j < edge.size(); j++)
    {
        if (edge[j].v == v && edge[j].u == u)
        {
            edge[j].flow -= flow;
            return;
        }
    }

    // adding reverse Edge in residual graph
    Edge e = Edge(0, flow, u, v);
    edge.push_back(e);
}

// To push flow from overflowing vertex u
bool Graph::push(int u)
{
    // Traverse through all edges to find an adjacent (of u)
    // to which flow can be pushed
    for (int i = 0; i < edge.size(); i++)
    {
```

```
            // if flow is equal to capacity then no push
            // is possible
            if (edge[i].flow == edge[i].capacity)
                continue;

            // Push is only possible if height of adjacent
            // is smaller than height of overflowing vertex
            if (ver[u].h > ver[edge[i].v].h)
            {
                // Flow to be pushed is equal to minimum of
                // remaining flow on edge and excess flow.
                int flow = min(edge[i].capacity - edge[i].flow,
                               ver[u].e_flow);

                // Reduce excess flow for overflowing vertex
                ver[u].e_flow -= flow;

                // Increase excess flow for adjacent
                ver[edge[i].v].e_flow += flow;

                // Add residual flow (With capacity 0 and negative
                // flow)
                edge[i].flow += flow;

                updateReverseEdgeFlow(i, flow);

                return true;
            }
        }
    }
    return false;
}

// function to relabel vertex u
void Graph::relabel(int u)
{
    // Initialize minimum height of an adjacent
    int mh = INT_MAX;

    // Find the adjacent with minimum height
    for (int i = 0; i < edge.size(); i++)
    {
        if (edge[i].u == u)
        {
            // if flow is equal to capacity then no
            // relabeling
            if (edge[i].flow == edge[i].capacity)
                continue;

            // Update minimum height
            if (ver[edge[i].v].h < mh)
            {
                mh = ver[edge[i].v].h;

                // updating height of u
                ver[u].h = mh + 1;
            }
        }
    }
}

// main function for printing maximum flow of graph
int Graph::getMaxFlow(int s, int t)
{
    preflow(s);
```

```
        int u = overFlowVertex(ver);
        if (!push(u))
            relabel(u);
    }

    // ver.back() returns last Vertex, whose
    // e_flow will be final maximum flow
    return ver.back().e_flow;
}

// Driver program to test above functions
int main()
{
    int V = 6;
    Graph g(V);

    // Creating above shown flow network
    g.addEdge(0, 1, 16);
    g.addEdge(0, 2, 13);
    g.addEdge(1, 2, 10);
    g.addEdge(2, 1, 4);
    g.addEdge(1, 3, 12);
    g.addEdge(2, 4, 14);
    g.addEdge(3, 2, 9);
    g.addEdge(3, 5, 20);
    g.addEdge(4, 3, 7);
    g.addEdge(4, 5, 4);

    // Initialize source and sink
    int s = 0, t = 5;

    cout << "Maximum flow is " << g.getMaxFlow(s, t);
    return 0;
}
```

Output

```
Maximum flow is 23
```

The code in this article is contributed by **Siddharth Lalwani** and **Utkarsh Trivedi**.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Recommended Posts:

Boruvka's algorithm for Minimum Spanning Tree
Push Relabel Algorithm | Set 1 (Introduction and Illustration)
Floyd Warshall Algorithm | DP-16
Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)
Kruskal's Minimum Spanning Tree Algorithm | Greedy Algo-2
Dijkstra's shortest path algorithm | Greedy Algo-7
Dijkstra's Algorithm for Adjacency List Representation | Greedy Algo-8
Bellman–Ford Algorithm | DP-23
Ford-Fulkerson Algorithm for Maximum Flow Problem

Graph Coloring | Set 2 (Greedy Algorithm)
Tarjan's Algorithm to find Strongly Connected Components
Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm)
Flood fill Algorithm - how to implement fill() in paint?

Article Tags : Graph

Practice Tags : Graph

Be the First to upvote.

To-do Done

4.1

Based on **7** vote(s)

Feedback/ Suggest Improvement Add Notes Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

11 Comments **GeeksforGeeks** 1 **Login**

Recommend Tweet Share **Sort by Newest**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Filip Masar • 6 days ago

Program does not stop on this graph when s = 2 and t = 1. Can someone tell me why ?

• Reply • Share ›

Filip Masar Filip Masar • 6 days ago • edited

I think I got it. overFlowVertex function should look like this:

```
// returns index of overflowing Vertex
int overFlowVertex(vector<vertex>& ver, int s, int t)
{
for (int i = 0; i < ver.size(); i++)
if (i != s && i != t && ver[i].e_flow > 0)
```

```
return -1;
}
```

and main function (int Graph::getMaxFlow(int s, int t)) should return

```
return ver[t].e_flow;
```

It works for me now. Correct me if I am wrong :-)

^ | v • Reply • Share ›

Vinay Chowdary • 10 months ago

Can anyone help me by providing the code for finding node-disjoitn paths between source to destination nodes

^ | v • Reply • Share ›

globalismcom • a year ago

How can the detailed flow (flow sent along each edge) can be obtained from this implementation?
It is straighforward to me, if there are no double arcs ((i,j) and (j,i), then the flow values for each edge can be easily reconstructed from the edge list, but in case there are double arcs things get confusing, as the residual graph is not stored seperately...

^ | v • Reply • Share ›

flashmozzg • a year ago

This algo completely ignores sink. It doesn't use \`t\` anywhere and return the same answer for any \`t\` you pass.

^ | v • Reply • Share ›

Rochak Saini • a year ago

Very precise explanation. Satisfied.

^ | v • Reply • Share ›

Aman Chauhan ➦ Rochak Saini • a year ago • edited

Amazed by the way things turned out in the end!

^ | v • Reply • Share ›

Rochak Saini ➦ Aman Chauhan • a year ago

That "return" statement almost took my breath away.

^ | v • Reply • Share ›

Show more replies

.oDaniel • 3 years ago

// adding reverse Edge in residual graph
Edge e = Edge(0, flow, u, v);

Should it be?
Edge e = Edge(-flow, 0, u, v);

^ | v • Reply • Share ›

Rango • 3 years ago

The given algorithm seems to be incorrect at some points.
Doesn't take into consideration the fact that push can only be done to vertices that are only 1 step downhill.
Also doesn't consider the fact that only those vertices could be relabeled that have all neighboring vertices either uphill or at same height.

^ | v • Reply • Share ›

Marcel Čampa ➦ Rango • 2 years ago • edited

You don't have to check for an adjacent vertex to be only one step downhill. Because if you can push to that vertex, it is

always one step downhill. That happens because you relabel height of the pushing vertex to be $1+\min(heights of adjacent vertices)$. Therefore you will only be able to push to adjacent vertex that has height smaller by 1. Otherwise they are the same height (or the adjacent is higher) and you need to do relabeling if there are no other adjacent vertices you could push to (either the edge between is saturated or the adjacent vertex has the same height (or greater)).
You don't need to check for that either. Lemma 26.14 from CLRS states, that "an overflowing vertex can be either pushed or relabel". You can find proof in the book. That means, that if you couldn't push (meaning the height of all adjacent vertices is greater), you have to relabel. To rephrase, it cannot happen that you will relabel a vertex when there exists an adjacent vertex with lower height.

• Reply • Share ›