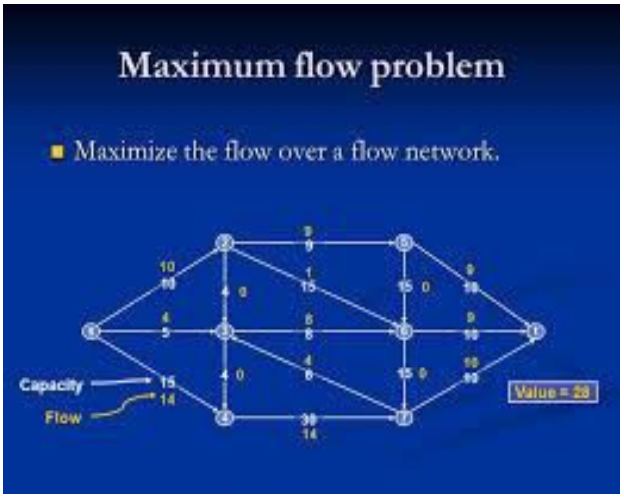


ALGORITHM DESIGN PROJECT

MAXIMUM FLOW PROBLEM

The project aims at implementing the existing algorithms to solve the problem and the analysis of the same. The main focus will be on naive greedy approach,Ford-Fulkerson algorithm, Edmonds–Karp algorithm and finally Dinic's algorithm.

Submitted by:

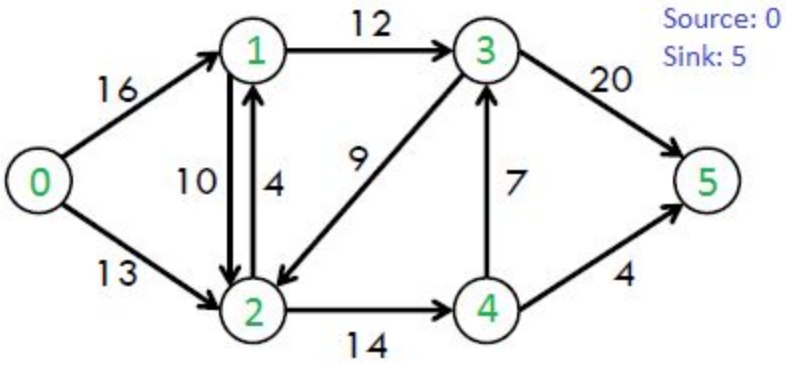
Anurag Verma 160001005
Aravind Ravikumar 160001006

Under the Guidance of

Dr. Kapil Ahuja

Problem Statement

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



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 :

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

Naive-Greedy Approach

Let's try the following approach. Initially set all weights in the flow graph to 0. Then:

1. Compute the edge weights in the residual graph Gr from the input capacity graph G and the current flow graph Gf. Gr shows us where there is unused flow capacity that we can possibly use. Consider 0-weight edges in Gr as nonexistent.

2. Consider simple paths from source s to sink t in Gr. These s-t paths are possible paths for pushing additional flow from s to t.
If there are no s-t paths in Gr, Done: Gf shows the maximum flow.
Each s-t path has a "bottleneck" edge. (The bottleneck edge on a path is the one that has the smallest edge weight -- i.e. capacity -- of any edge on that path.)
Find the s-t path P with the largest bottleneck edge weight. Call this edge weight b. P is an augmenting path P that will permit increasing flow from s to t by amount b.

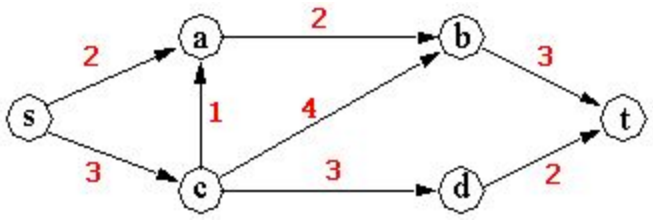
3. Update the flow graph Gf with the additional flow along the augmenting path P:

Consider the edges in the augmenting path P, in Gf . Add P's "bottleneck" flow b to the current weight of each of those edges in the flow graph Gf .

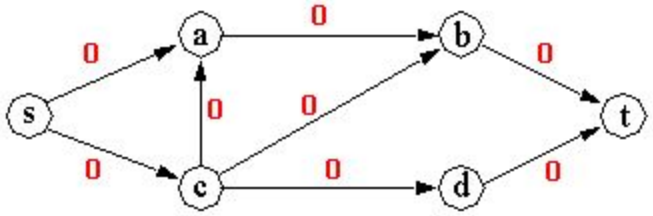
4. Go to 1.

Example of greedy approach:

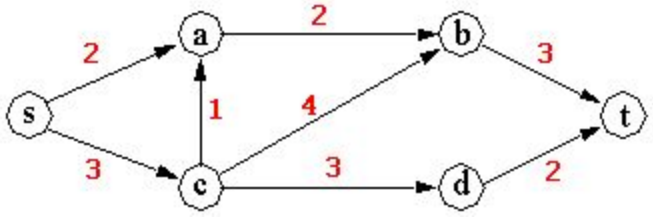
Input capacity graph G:



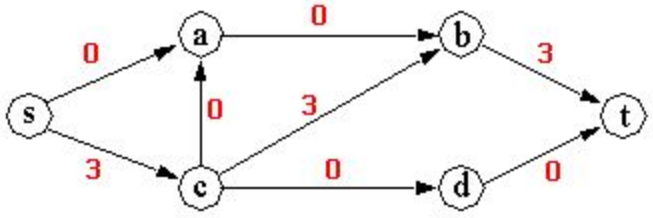
A flow graph Gf , initially all weights 0:



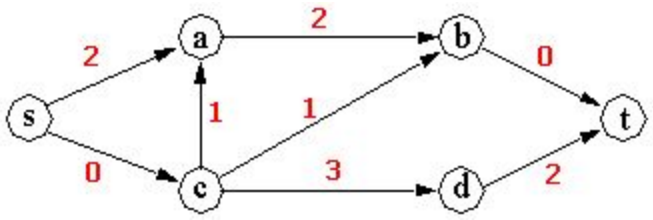
Residual graph Gr:, first max-bottleneck s-t path is (s,c),(c,b),(b,t), bottleneck 3...



Update flow graph Gf, adding bottleneck weight along augmenting path



Update residual graph Gr ... but now there is no augmenting path... done



Fixing the greedy maximum-flow algorithm

That simple greedy approach did not work.
Evidently, we need some way of backtracking: being able to undo some flow on some edges, if we overcommitted and used `too much' flow there. Basic idea for how to do this, leaving everything else in the algorithm unchanged. When computing the residual graph given the current flow graph, add some additional edges but, these additional edges will have the same weight as edges in the flow graph, but that point in the opposite direction, let's call these additional edges backward edges.
Backward edges, as well as the regular forward edges, can be included in an augmenting path in the residual graph. However, when using the augmenting path bottleneck weight to update the flow graph weights, if an augmenting path edge is a backward edge, its weight gets subtracted from the current flow graph edge (undoing flow), instead of added.
It turns out that this idea works, and leads to the Ford-Fulkerson algorithm

Ford-Fulkerson Algorithm

The Ford–Fulkerson algorithm (FFA) is a greedy algorithm that computes the maximum flow in a flow network.The idea behind the algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

Initially set all weights in the flow graph to 0. Then:

1. Compute the edge weights in the residual graph Gr from the current flow graph Gf :
For each edge (u,v,c) in G, and corresponding edge (u,v,f) in Gf , do the following:
Update forward edge (u,v,x) in Gr to be (u,v, c-f)
If f>0, create or update backward edge (v,u,x) in Gr to be (v,u, f)
Consider 0-weight edges in the resulting Gr as nonexistent.

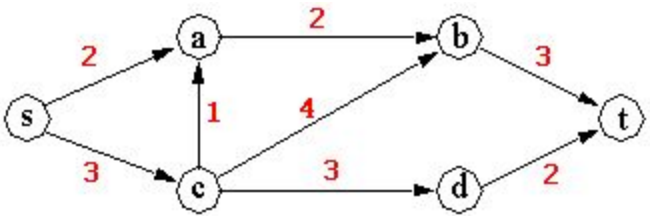
2. Find the s-t augmenting path P in Gr with largest bottleneck edge weight b. If there is no s-t path in Gr , Done: Gf shows the maximum flow.

3. Update the flow graph Gf with the flow along the augmenting path P:
For each edge (u,v,x) in P, update edge weights in the flow graph Gf :
If (u,v) is a forward edge in Gr , update edge (u,v,f) in Gf to be (u,v, f+b).
If (u,v) is a backward edge in Gr , update edge (v,u,f) in Gf to be (v,u, f-b).

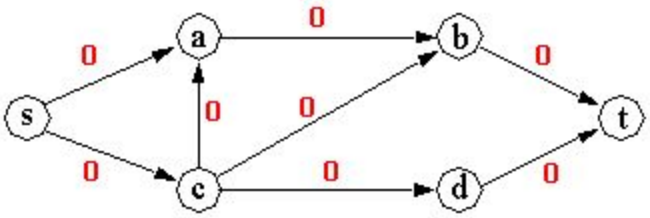
4. Go to 1.

Examples of Ford-Fulkerson:

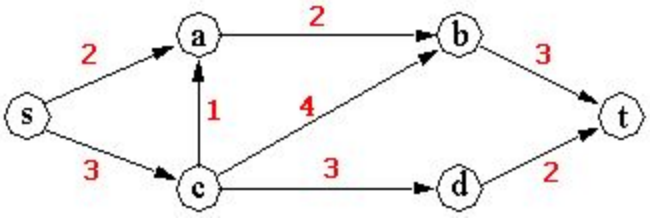
Input capacity graph G:



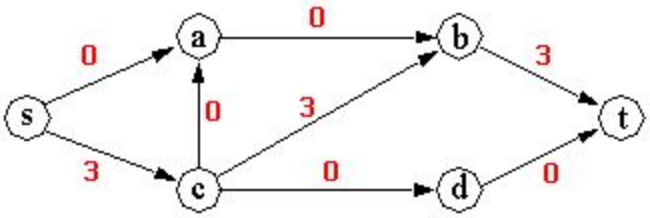
A flow graph Gf , initially all weights 0:



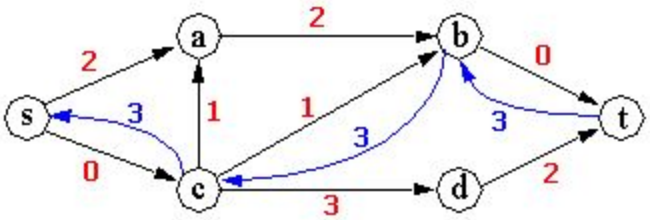
Residual graph Gr:, first max-bottleneck s-t path is (s,c),(c,b),(b,t), bottleneck 3



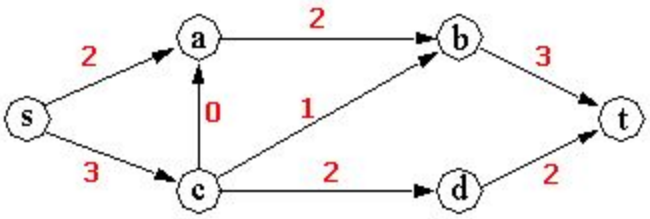
Update flow graph Gf, adding bottleneck weight along augmenting path



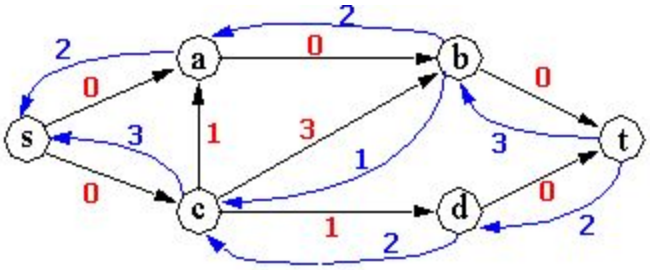
Update residual graph Gr with fwd+bkwd edges, and find best augmenting path



Update flow graph Gf, adding/subtracting bottleneck weight along augmenting path



Update residual graph Gr with fwd+bkwd edges, and... no augmenting path, done



Implementation

```
//Implementation of Ford-Fulkerson algorithm using dfs to find
//augmenting path and not max bottleneck as shown above
//code framework http://wikistack.com/max-flow-problem-ford-fulkerson-algorithm/

#include <iostream>
#include <string.h>
using namespace std;

#define N 7
#define INF 9999999

// flow network
int Flow[N][N];
// visited array
bool visited[N];

// original flow network graph shown in the above example
```

```
        //0 1  2  3  4 5  6
int graph[N][N] = {
                { 0, 5, 4, 0, 0, 0, 0 }, //0
                { 0, 0, 0, 0, 0, 0, 4 }, //1
                { 0, 0, 0, 3, 0, 0, 6 }, //2
                { 0, 0, 0, 0, 5, 0, 0 }, //3
                { 0, 0, 0, 0, 0, 0, 8 }, //4
                { 6, 0, 0, 2, 0, 0, 0 }, //5
                { 0, 0, 0, 0, 0, 0, 0 }, //6
                };

int dfs(int s, int t, int minimum) {
        visited[s] = true;

        // if source and sink is same
        if (s == t)
                return minimum;

        for (int i = 0; i < N; i++) {
                int flow_capacity = graph[s][i] - Flow[s][i];
                if (!visited[i] && flow_capacity > 0) {
                        // find min capacity in dfs path
                        if (int sent = dfs (i, t, min (minimum, flow_capacity))) {
                                // adjust the capacity
                                Flow[s][i] += sent;
                                Flow[i][s] -= sent;
                                return sent;
                        }
                }
        }

        return false;
}

int main() {
        // initialize initial flow capacity 0
        memset(Flow, 0, sizeof(Flow));

        // initialize visited array false initially
        memset(visited, 0, sizeof(visited));

        int s = 5;
        int t = 6;
```

```
        int max_flow = 0;
        // while ther is augmenting path , from s and t
        // with positive flow capacity
        while (int sent = dfs(s, t, INF)) {
                max_flow += sent;
                // reset visited array , for searching next path
                memset(visited, 0, sizeof(visited));
        }
        cout << "The max flow from node 5 to sink node 6 is " << max_flow;
        cout << endl;
}
//complexity of the above implementation is O(E max|f|)
```

Edmond-Karp Algorithm

The algorithm is identical to the Ford–Fulkerson algorithm, except that the search order when finding the augmenting path is defined. The path found must be a shortest path that has available capacity. This can be found by a breadth-first search, where we apply a weight of 1 to each edge.

The variation is in step 2. Insead of finding the s-t path with the largest bottleneck, just find the s-t path with the fewest edges:

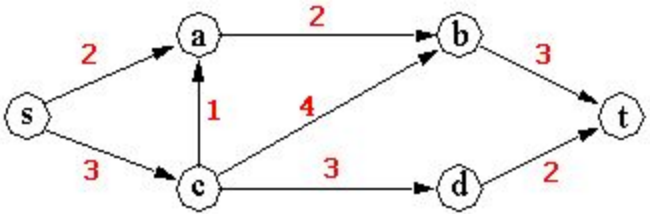
2. Find the s-t augmenting path P in Gr with the fewest edges. Let its bottleneck edge weight be b. If there is no s-t path in Gr , Done: Gf shows the maximum flow.

Since the path with the fewest edges can easily be found with breadth-first search, this variation is simple to implement

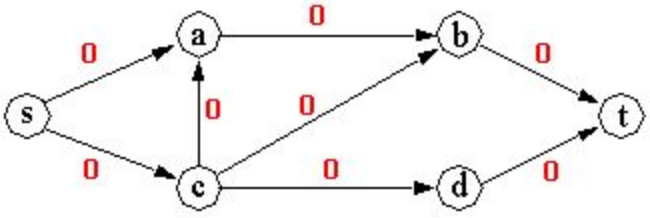
This just looks like a simple heuristic: short paths are maybe less likely to have restrictive bottlenecks, just because they have fewer edges.

Examples of Edmond-Karp:

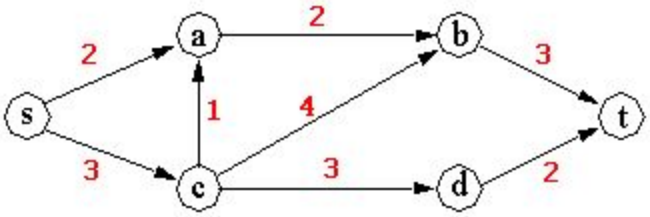
Input capacity graph G:



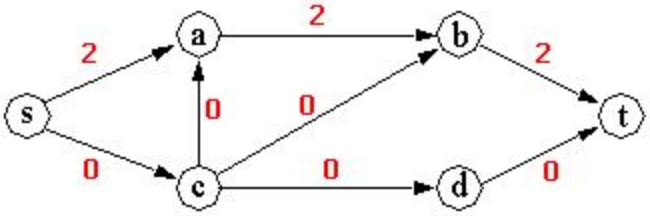
A flow graph Gf , initially all weights 0



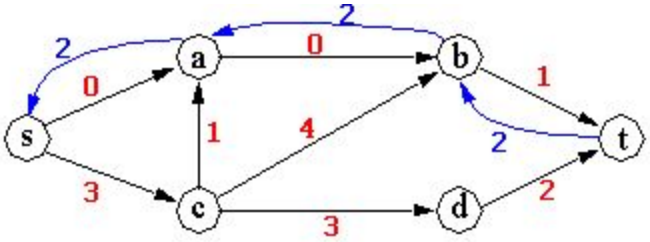
Residual graph Gr:, a shortest s-t path is (s,a),(a,b),(b,t), bottleneck 2...



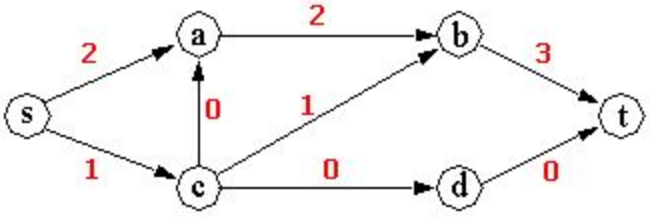
Update flow graph Gf, adding bottleneck weight along augmenting path



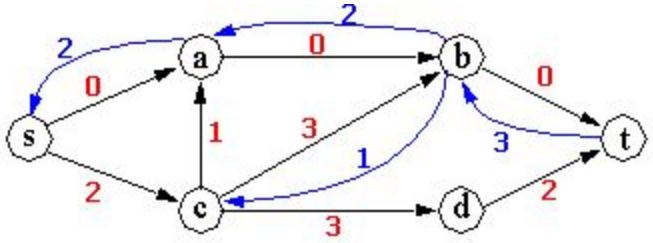
Update residual graph Gr with fwd+bkwd edges, and find a shortest s-t path



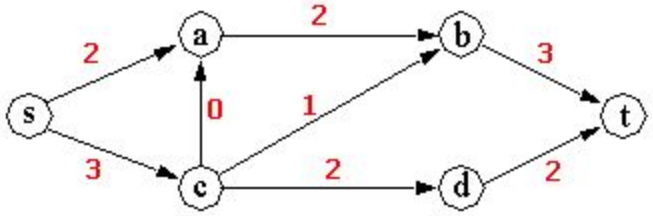
Update flow graph Gf, adding bottleneck weight along augmenting path



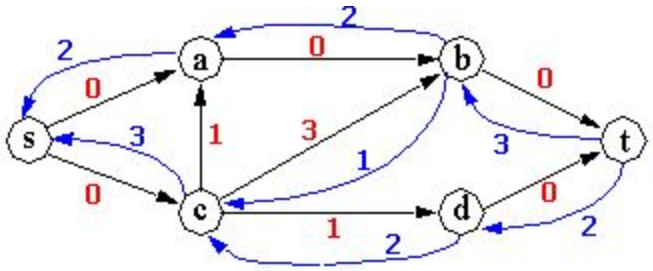
Update residual graph Gr with fwd+bkwd edges, and find shortest s-t path



Update flow graph Gf, adding/subtracting bottleneck weight along augmenting path



Update residual graph Gr with fwd+bkwd edges, and... no augmenting path, done



Implementation

```
// C++ program for implementation of Edmond-Karp algorithm
//similar to the previous implementation, except that it uses bfs instead of dfs
//code framework from
//https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

#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 sink 't' in
  residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // 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, then return
    // true, else false
    return (visited[t] == true);
}

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

```
    // Create a residual graph and fill the residual graph with
    // given capacities in the original graph as residual capacities
    // in residual graph
    int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
                      // residual capacity of edge from i to j (if there
                      // is an edge. If rGraph[i][j] is 0, then there is not)
    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 and to store path

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

    // Augment the flow while tere is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edges along the
        // path filled by BFS. Or we can say find the maximum flow
        // 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 reverse edges
        // 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 example
    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 " << fordFulkerson(graph, 0, 5);

    return 0;
}
```

//complexity of the above implementation is $O(VE^2)$

Dinic’s Blocking Flow Algorithm

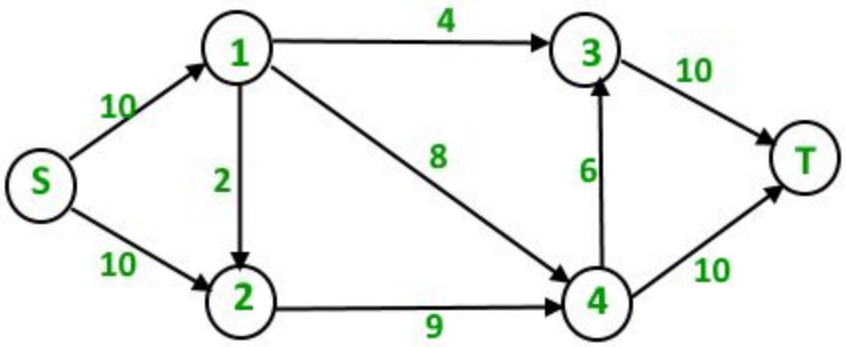
Dinic's algorithm is a strongly polynomial algorithm for computing the maximum flow in a flow network, conceived in 1970 by Israeli (formerly Soviet) computer scientist Yefim (Chaim) A. Dinitz. The introduction of the concepts of the level graph and blocking flow enable Dinic's algorithm to achieve its performance.

1) Initialize residual graph G as given graph.

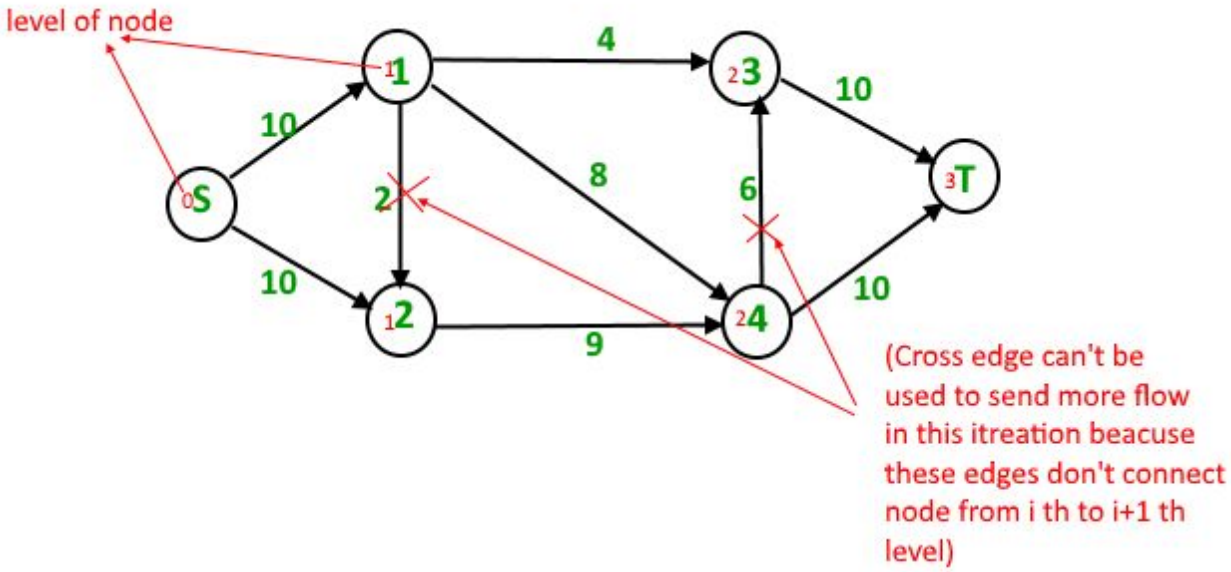
2) Do BFS of G to construct a level graph (or assign levels to vertices) and also check if more flow is possible.
   a) If more flow is not possible, then return.
   b) Send multiple flows in G using level graph until blocking flow is reached. Here using level graph means, in every flow, levels of path nodes should be 0, 1, 2... (in order) from s to t.

Example of Dinic's algorithm

Initial Residual Graph

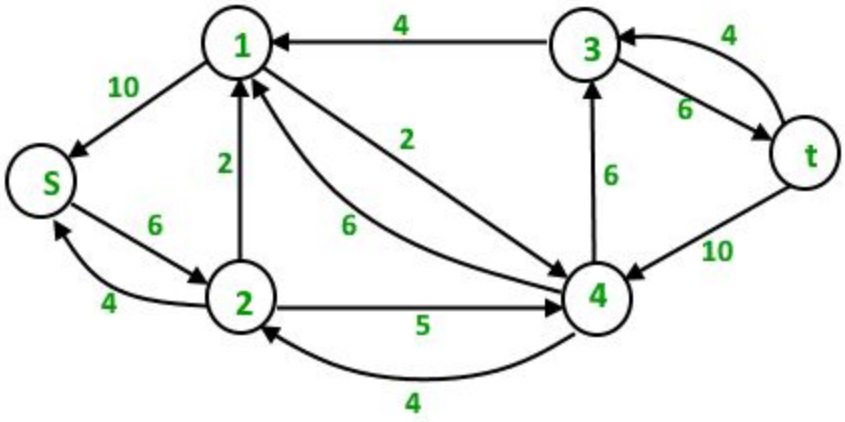


Total Flow = 0
First Iteration : We assign levels to all nodes using BFS. We also check if more flow is possible (or there is a s-t path in residual graph).

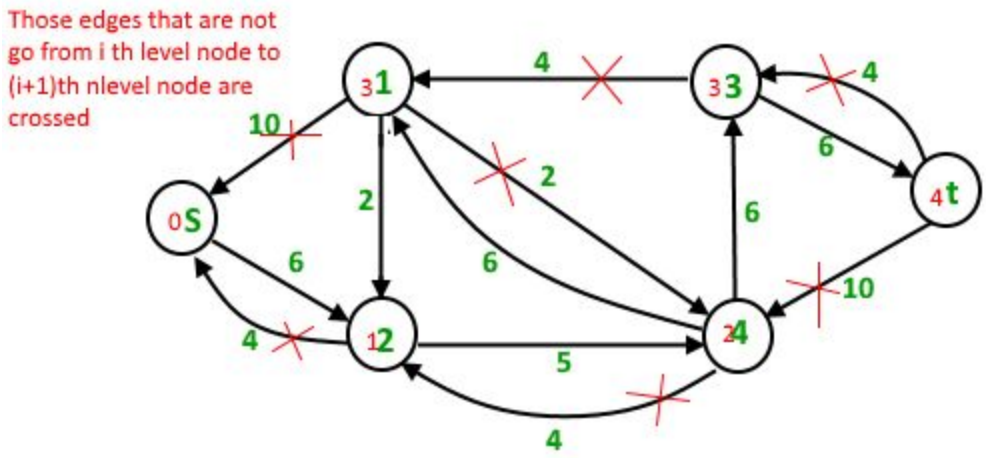


Now we find blocking flow using levels (means every flow path should have levels as 0, 1, 2, 3). We send three flows together. This is where it is optimized compared to Edmond Karp where we send one flow at a time.
4 units of flow on path s – 1 – 3 – t.
6 units of flow on path s – 1 – 4 – t.
4 units of flow on path s – 2 – 4 – t.
Total flow = Total flow + 4 + 6 + 4 = 14

After one iteration, residual graph changes to following.



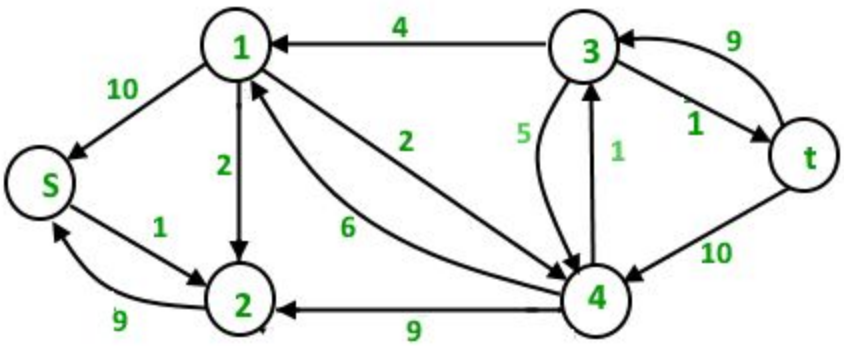
Second Iteration : We assign new levels to all nodes using BFS of above modified residual graph. We also check if more flow is possible (or there is a s-t path in residual graph).



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

```
// C++ implementation of Dinic's Algorithm
// code framework https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/

#include<bits/stdc++.h>
using namespace std;

// A structure to represent a edge between
// two vertex
struct Edge
{
    int v ;  // Vertex v (or "to" vertex)
             // of a directed edge u-v. "From"
             // vertex u can be obtained using
             // index in adjacent array.

    int flow ; // flow of data in edge

    int C;    // capacity

    int rev ; // To store index of reverse
              // edge in adjacency list so that
              // we can quickly find it.
};
```

```
// Residual Graph
class Graph
{
    int V; // number of vertex
    int *level ; // stores level of a node
    vector< Edge > *adj;
public :
    Graph(int V)
    {
        adj = new vector<Edge>[V];
        this->V = V;
        level = new int[V];
    }

    // add edge to the graph
    void addEdge(int u, int v, int C)
    {
        // Forward edge : 0 flow and C capacity
        Edge a{v, 0, C, adj[v].size()};

        // Back edge : 0 flow and 0 capacity
        Edge b{u, 0, 0, adj[u].size()};

        adj[u].push_back(a);
        adj[v].push_back(b); // reverse edge
    }

    bool BFS(int s, int t);
    int sendFlow(int s, int flow, int t, int ptr[]);
    int DinicMaxflow(int s, int t);
};

// Finds if more flow can be sent from s to t.
// Also assigns levels to nodes.
bool Graph::BFS(int s, int t)
{
    for (int i = 0 ; i < V ; i++)
        level[i] = -1;

    level[s] = 0;  // Level of source vertex

    // Create a queue, enqueue source vertex
    // and mark source vertex as visited here
```

```
    // level[] array works as visited array also.
    list< int > q;
    q.push_back(s);

    vector<Edge>::iterator i ;
    while (!q.empty())
    {
        int u = q.front();
        q.pop_front();
        for (i = adj[u].begin(); i != adj[u].end(); i++)
        {
            Edge &e = *i;
            if (level[e.v] < 0  && e.flow < e.C)
            {
                // Level of current vertex is,
                // level of parent + 1
                level[e.v] = level[u] + 1;

                q.push_back(e.v);
            }
        }
    }

    // IF we can not reach to the sink we
    // return false else true
    return level[t] < 0 ? false : true ;
}

// A  DFS based function to send flow after BFS has
// figured out that there is a possible flow and
// constructed levels. This function called multiple
// times for a single call of BFS.
// flow : Current flow send by parent function call
// start[] : To keep track of next edge to be explored.
//           start[i] stores  count of edges explored
//           from i.
//  u : Current vertex
//  t : Sink
int Graph::sendFlow(int u, int flow, int t, int start[])
{
    // Sink reached
    if (u == t)
        return flow;
```

```
    // Traverse all adjacent edges one -by - one.
    for (  ; start[u] < adj[u].size(); start[u]++)
    {
        // Pick next edge from adjacency list of u
        Edge &e = adj[u][start[u]];

        if (level[e.v] == level[u]+1 && e.flow < e.C)
        {
            // find minimum flow from u to t
            int curr_flow = min(flow, e.C - e.flow);

            int temp_flow = sendFlow(e.v, curr_flow, t, start);

            // flow is greater than zero
            if (temp_flow > 0)
            {
                // add flow  to current edge
                e.flow += temp_flow;

                // subtract flow from reverse edge
                // of current edge
                adj[e.v][e.rev].flow -= temp_flow;
                return temp_flow;
            }
        }
    }

    return 0;
}

// Returns maximum flow in graph
int Graph::DinicMaxflow(int s, int t)
{
    // Corner case
    if (s == t)
        return -1;

    int total = 0;  // Initialize result

    // Augment the flow while there is path
    // from source to sink
    while (BFS(s, t) == true)
```

```
    {
        // store how many edges are visited
        // from V { 0 to V }
        int *start = new int[V+1];

        // while flow is not zero in graph from S to D
        while (int flow = sendFlow(s, INT_MAX, t, start))

            // Add path flow to overall flow
            total += flow;
    }

    // return maximum flow
    return total;
}

// Driver program to test above functions
int main()
{
    Graph g(6);
    g.addEdge(0, 1, 16 );
    g.addEdge(0, 2, 13 );
    g.addEdge(1, 2, 10 );
    g.addEdge(1, 3, 12 );
    g.addEdge(2, 1, 4 );
    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);

    // next exmp
    /*g.addEdge(0, 1, 3 );
      g.addEdge(0, 2, 7 ) ;
      g.addEdge(1, 3, 9);
      g.addEdge(1, 4, 9 );
      g.addEdge(2, 1, 9 );
      g.addEdge(2, 4, 9);
      g.addEdge(2, 5, 4);
      g.addEdge(3, 5, 3);
      g.addEdge(4, 5, 7 );
      g.addEdge(0, 4, 10);
```

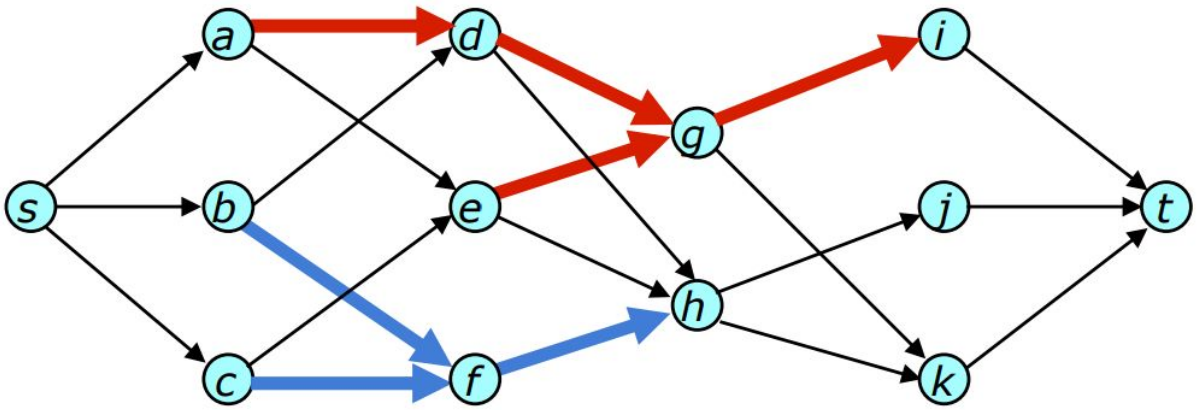
```
    // next exp
    g.addEdge(0, 1, 10);
    g.addEdge(0, 2, 10);
    g.addEdge(1, 3, 4 );
    g.addEdge(1, 4, 8 );
    g.addEdge(1, 2, 2 );
    g.addEdge(2, 4, 9 );
    g.addEdge(3, 5, 10 );
    g.addEdge(4, 3, 6 );
    g.addEdge(4, 5, 10 ); */

   cout << "Maximum flow " << g.DinicMaxflow(0, 5);
   return 0;
}
```

// complexity of above implementation is $O(V^2E)$

Dinic's Algorithm with Dynamic Trees

The primary source of inefficiency in Dinic's algorithm is that successive augmenting searches may re-discover paths with positive residual capacity that were already found in previous searches. Instead of starting each search from scratch, we could remember portions of the graph that still have positive residual capacity. Thus, instead of searching for an augmenting path edge-by-edge, we could try to combine existing path segments to produce the desired path.



The above figure shows two directed subtrees of the residual graph, one with 1 root i and one with root h. These subtrees consist of edges (u, v) in the residual graph with res(u, v) > 0 and level(v) = level(u) + 1. Suppose we start a path search from s using the edge (a, s). Because a is part of the subtree with root i, we can use this information to avoid explicitly searching the edges on the path from a to i. We'll see that with the appropriate representation of the subtrees, we can jump to the root of a subtree in O(log n) time, potentially reducing the time needed for path searches by a factor of O(n/ log n). The dynamic trees data structure represents a collection of trees on n vertices, where each vertex has an associated cost. The data structure idea is as follows.

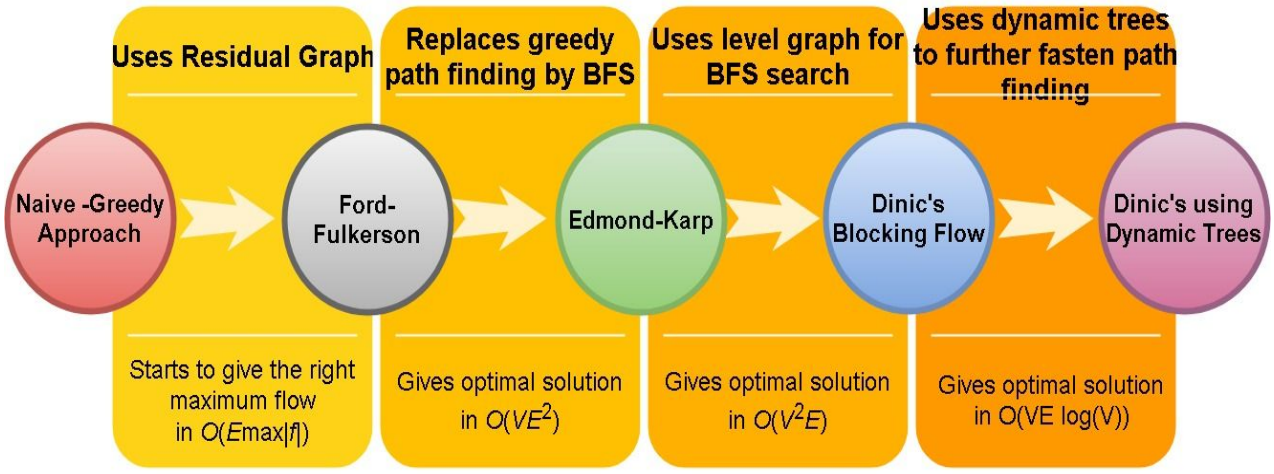
"Partial paths" during DFS form a forest of trees, so store them as a dynamic tree. Overview:
1. Start with each node in its own tree
2. Define a "current node" v which will always be the node at the end of the path.
3. Set v equal to root(s).
4. If v != t, then we want to move forward one edge in the DFS. If this edge is (v, w), then we perform link(v, w) in the dynamic tree, linking them with an edge of the same weight as (v, w), and start again from step 3.
5. If there were no edges coming out from v, then we want to delete it, so we delete all incoming edges (u, v). If any of these are also edges in the dynamic tree, perform cut(u).
6. If v = t, then we have a path from s to t in the dynamic tree. Use the minweight and update operations to perform the augmentation in the dynamic tree, and then delete all edges along the resulting path with weight 0 and cut them.
7. If there are no edges coming out of s, then we're done.

When we perform an augmentation in the dynamic tree, we don't actually know what the specific path is. However, we can still compute the overall blocking flow by recording the weight on each edge as it gets deleted from the dynamic tree. O(V) operations because at most O(V) edge deletions and augmentations, so total time O(E log V) to find a blocking flow and overall running time O(VE log V).

Comparing Complexities

Linear programming		Constraints given by the definition of a legal flow. See the linear program here.

Ford–Fulkerson algorithm	$O(E \max\|f\|)$	As long as there is an open path through the residual graph, send the minimum of the residual capacities on the path. The algorithm is only guaranteed to terminate if all weights are rational. Otherwise it is possible that the algorithm will not converge to the maximum value. However, if the algorithm terminates, it is guaranteed to find the maximum value.
Edmonds–Karp algorithm	$O(VE^2)$	A specialization of Ford–Fulkerson, finding augmenting paths with breadth-first search.
Dinic's blocking flow algorithm	$O(V^2E)$	In each phase the algorithms builds a layered graph with breadth-first search on the residual graph. The maximum flow in a layered graph can be calculated in O(VE) time, and the maximum number of the phases is n-1.
Dinic's algorithm	$O(VE \log(V))$	The dynamic trees data structure speeds up the maximum flow computation in the layered graph to $O(\mathrm{V}\ E \log(V))$.

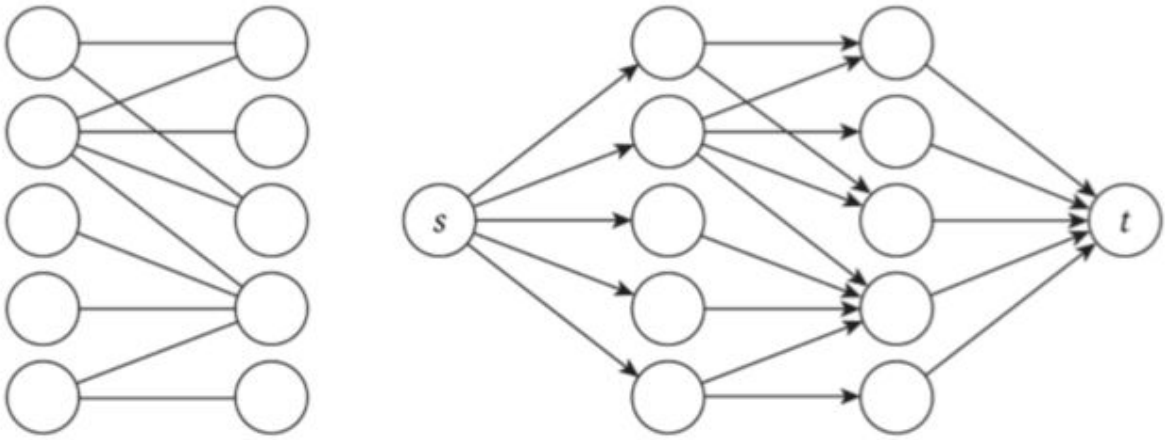


Applications of Network Flow

Some of the popular applications of the network flow are:

1.Bipartite Matching:

Problem Statement: Given an instance of a bipartite graph G. Find the matching of Largest size in G.



Algorithm:

1. Convert G to network flow G'. Let X and Y be the two parts of the Bipartite graph. Then direct edges from X to Y, add nodes s and t, connect s to each node in X, connect each node in Y to t. Make the capacity of each edge '1'.
2. Compute the maximum flow in G'.

Claim: This maximum flow will be the size of maximum matching of the given bipartite graph.

Proof of Claim:

1. Matching → flow: if there is a matching with k edges in G, there is an s-t flow of value k in G'.
2. Flow → matching: if there is an integer-valued flow f0 in G0 with value k, there is a matching M in G with k edges.
    A. There is an integer-valued flow f of value k ⇒ flow along any edge is 0 or 1.
    B. Let M be the set of edges not incident on s or t with flow equal to 1.
    C. Claim: M contains k edges.
    D. Claim: Each node in X (respectively, Y) is the tail (respectively, head) of at most one edge in M.

3. Conclusion: size of the maximum matching in G is equal to the value of the maximum flow in G0; the edges in this matching are those that carry flow from X to Y in G0.

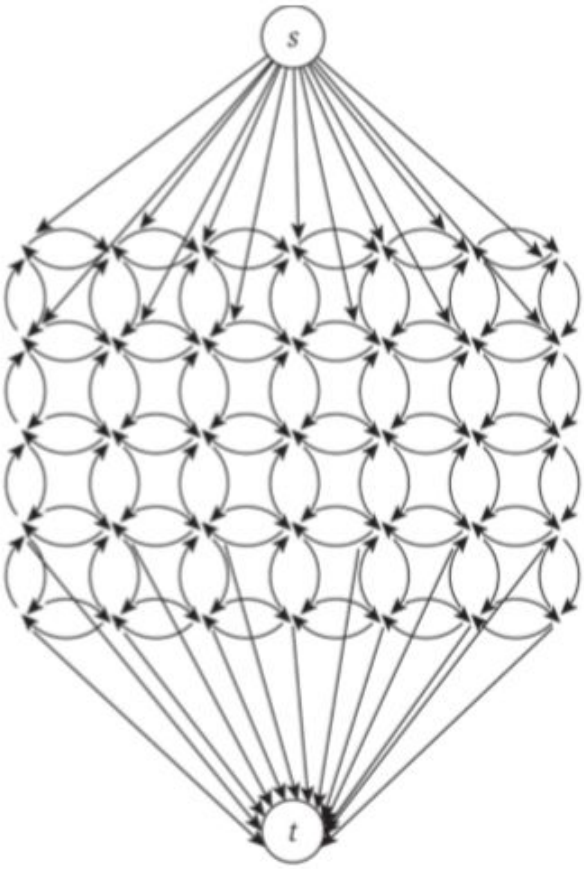
2. Image Segmentation:

Problem Statement: Given an instance of Pixel Graphs G(V,E); the likelihood functions a, b:V-> R+ and the penalty function p: E->R+. Partition the pixels into two sets A and B that maximises:

$$q(A,B) = \sum_{i \in A0} a_i + \sum_{j \in B} b_j - \sum_{((i,j) \in E;\ |A \cap \{i,j\}|=1)} p_{ij}$$

Algorithm:

1. Add a new 'super-source' s and 'super-sink' t to represent the foreground and background respectively.
2. Connect s and t to every pixel and assign capacity ai to edge(s,i) and capacity bi to edge (i,t).
3. Direct edges away from s and t.
4. Replace edge (i,j) in E with two directed edges of capacity pij.
5. To maximize q(A,B) compute the s-t cut (A,B) of minimum capacity.



3. Baseball Elimination Problem:

Problem Statement: There are n teams competing in a league. At a specific stage of the league season, w_i is the number of wins and r_i is the number of games left to play for team i and r_{ij} is the number of games left against team j. A team is eliminated if it has no chance to finish the season in the first place.

Algorithm:
Let $G = (V, E)$ be a network with $s,t \in V$ being the source and the sink respectively. One adds a game node $\{i,j\}$ with $i < j$ to V, and connects each of them from s by an edge with capacity r_{ij} – which represents the number of plays between these two teams. We also add a team node for each team and connect each game node $\{i,j\}$ with two team nodes i and j to ensure one of them wins. Finally, edges are made from team node i to the sink node t and the capacity of $w_k+r_k-w_i$ is set to prevent team i from winning more than w_k+r_k. Let S be the set of all teams participating in the league and let

$$r(S - \{k\}) = \sum i, j \in \{S - \{k\}\}, i < j \; r \; i \; j.$$

In this method it is claimed team k is not eliminated if and only if a flow value of size $r(S - \{k\})$ exists in network G.

Recent Researches in Network Flow

The first maximum flow algorithms, the network simplex method and the augmenting path method , were developed in the 50's. These algorithms are pseudo polynomial. In the early 70's, Dinitz and Edmonds and Karp showed that the shortest augmenting path algorithm is polynomial. The main idea of this algorithm is to assign unit lengths to residual arcs and augment flow along a shortest augmenting path. One can show that the source to sink distance is nondecreasing and must increase after polynomially many augmentations.

During the next decade, maximum flow algorithms became faster due to the discovery of powerful new techniques. Dinitz developed the blocking flow method in the context of the shortest augmenting path algorithm; Karzanov [36] stated blocking flows as a separate problem and suggested the use of preflows to solve it. Edmonds and Karp and independently Dinitz developed capacity scaling. Dinitz also shows how to combine blocking flows and capacity scaling to obtain an O**(nm) maximum flow algorithm. Galil and Naamad developed data structures to speed up flow computations and obtained an O* (nm) algorithm. We use E instead of O when stating expected time bounds. For the unit capacity case, Karzanov and independently Even and Tarjan show that Dinitz' blocking flow algorithm solves the maximum flow problem in O(m 3/2) time on multigraphs and in O(min(n 2/3, ml/2)m) time on simple graphs. The push-relabel method, first used and fully developed by Goldberg and Tarjan, leads to better theoretical understanding of the problem and substantial improvements in

computational performance of the maximum flow codes . However, for a long time all bound improvements were limited to logarithmic factors.

Recently, polynomial improvements have been made for two classes of problems: undirected unit capacity problems and integer capacity problems. We discuss these cases below, starting with the latter.

Integer Capacity Flows

Dinitz' 1973 algorithm was the first one to achieve an O**(nm) bound. For a long time, the O(nm) bound was the goal of much of the theoretical research in the area. Recently, however, Goldberg and Rao developed an O** (min(n 2/3, ml/2)m) algorithm, bridging the gap between the unit and integer capacity cases. This result suggests that the O(nm) goal may be too modest. The Goldberg-Rao algorithm is a generalization of the blocking flow method that uses a non-unit length function. Recall that the main loop of the blocking flow method finds a blocking flow in an acyclic graph. Such a flow can be found in O(m log n) time . Recall that the original blocking flow method assigns unit lengths to all residual arcs. The use of non-unit length functions for maximum flow computations has been first suggested by Edmonds and Karp. In the context of minimum-cost flows, the idea has been studied in. However, prior to the Goldberg-Rao work, non-unit length functions never gave asymptotic time bound improvements. Goldberg and Rao use an adaptive binary length function.1 The length of a residual arc depends on the estimated value F of the residual flow. Arcs whose residual capacity is small compared to F have zero lengths and arcs whose residual capacity is large have unit lengths. Zero-length arcs, which seem essential for the algorithm, introduce technical complications, resulting from the fact that the graph G J induced by the residual arcs on shortest paths to the sink may not be acyclic. To deal with this complication, the algorithm contracts strongly connected components of G J. The algorithm limits the amount of flow through each contracted node so that it is easy to route the flow inside the nodes. Because the zero-length arcs have large capacity, the limit is large, and the algorithm makes significant progress even if the limit comes into play.

Here is a brief outline of the algorithm. We denote min(n 2/3, m U2) by A. The threshold between length one and length zero arcs is O(F/A).

The algorithm maintains a flow and a cut. The residual capacity of this cut gives an upper bound on the residual flow value. The algorithm also maintains a parameter F, which is an upper bound on the residual flow value. When the algorithm finds a cut with residual capacity of at most F/2, it sets F to this residual capacity, thereby decreasing F by at least a factor of two. Distances to the sink are monotone while F does not change. At each iteration, dominated by a blocking flow computation on the contracted graph G', either the flow value increases by at least F/A or the source to sink distance increases by at least one. One can show that when the distance exceeds cA, for a fixed constant c, F must decrease. This after O(A) iterations, F must decrease, either because the ftow value increased by at least F/2 or because the distance between the source and the sink exceeded cA. This leads to the desired time bound.

Undirected Unit Capacity Flows

In the undirected case, the unit capacity flow bound of O(min(n 2/3, mi/2)m) can be improved using sparsification techniques. Two different classes of sparcification techniques, originally developed for the minimum cut problem, proved useful for the maximum flow problem: sparse certificates and random sampling. The former was developed by Nagamochi and Ibaraki . The latter was developed by Sarger. In this section we assume that the input graph is undirected and all edge capacities are one and all distances are for the unit length function on the residual graph. For simplicity we assume that the input graph is simple and state the bounds in terms of the graph size. For the bounds depending on the flow value and the results for multigraphs, . The O(min(n 2/a, mW2)m) bound on Dinitz' blocking flow algorithm for the unit capacity case is based on the following two theorems.

Theorem 1. Each iteration of the blocking flow algorithm increases the distance between the source and the sink.

Theorem 2.If the distance between the source and the sink is d, then the residual flow value is O(min(m/d, (n/d)2)).

What Next?

One can see that significant progress has been made for the integral and the undirected unit capacity cases. The corresponding algorithms use powerful new design and analysis techniques. Better understanding and refinement of these techniques may lead to even faster algorithms. Although logarithmic factor improvements in time bounds are more likely, polynomial improvements may be possible as well, in particular in the undirected unit capacity case.

Still there is polynomial gap between the integral and the unit capacity cases. A result of Orlin for minimum-cost flows gives some hope that it may be possible to close this gap. Another gap is between the undirected and the directed unit capacity cases. Extending the undirected case results to the directed case requires sparcification techniques for directed graphs. This is a very challenging open problem. Similarly, closing the gap between the unit and the integral undirected cases seems to require directed graph sparcification because of the reduction to the integral directed case.

What is a natural limit of the current techniques? Suppose we have ideal sparsification: in O(m) time, we reduce the number of arcs in the graph we work on to O(n). Furthermore suppose we can close the gaps between the directed and undirected and the integral and real capacity cases. Then we would be able to solve the maximum flow problem in O* (m + n 3/2) time. This ambitious goal may be feasible: Goldberg and Rao show that random sampling of [6] can be used to find a cut of capacity within (1 + e) factor of the minimum in O*(m + n 3/2) time, for any constant e > 0. (A related result, due to Karger , is that a flow of value within (1 - e) of the maximum can be found in O*(mvrn/e) time.) We conclude by stating the following open question. Can the maximum flow problem in sparse unit capacity networks be solved in O(W,

3/2) time? A polynomial improvement to the O(rt 3/2) bound for this problem would be a real breakthrough.

Citations

1. Wikipedia : https://en.wikipedia.org/wiki/Maximum_flow_problem
2. Geeksforgeeks : https://www.geeksforgeeks.org/max-flow-problem-introduction/
3. http://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec12.netflow/lec12.html
4. Hackerearth:https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/tutorial/
5. https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/07NetworkFlowI.pdf
6. Stack Overflow: https://stackoverflow.com/
7. https://link.springer.com/content/pdf/10.1007/BFb0054350.pdf