Maximum Flow Problem: Ford-Fulkerson Method and Its Implementation

Kevin Angelo 13517086
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517086@std.stei.itb.ac.id

***Abstract*—Algorithms have experienced such improvements since its existence centuries ago; be it mathematical techniques like the Euclidean algorithm in computing the greatest common divisor of two numbers, or even algorithms in graph theories like Dijkstra's shortest path algorithm. This paper will mainly cover one among the many outstanding algorithms we have now; Ford-Fulkerson method and its implementation, the Edmonds-Karp algorithm. This algorithm basically computes the maximum flow that can be passed through a directed graph-like system, aiming for efficiency and optimal utilization of the edges of said graph.**

***Keywords*—algorithms, maximum flow problem, Ford-Fulkerson, Edmonds-Karp, graph theory, greedy, bfs**

I. Introduction

Ford-Fulkerson method or Ford-Fulkerson algorithm is a greedy algorithm in computing the maximum flow in a flow network. It is generally called a 'method' instead of an 'algorithm' because the full implementation of finding the augmenting path in the graph is not fully specified in their published papers. Maximum flow problem can be considered as a special case of more complex network problems, all of them involving directed graphs with the flow starting from a starting node 's' and ending in a terminal node 't'.

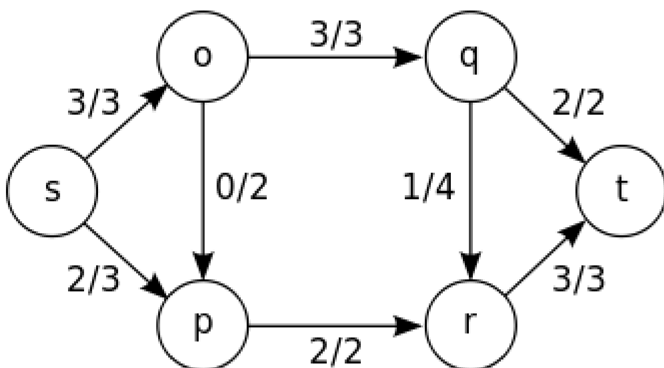


Figure 1. Directed graphs in maximum flow problem
(https://en.wikipedia.org/wiki/File:Max_flow.svg, accessed on April 24, 2019 22:17 GMT+7)

The maximum flow problem was first formulated in 1954 by T.E. Harris and F.S. Ross as a simplified model of Soviet railway traffic flow when they were doing their research for a secret documentation for the US Air Force. In attempts of solving the problem, Lester Randolph Ford Jr. and Delbert Ray Fulkerson created the first known algorithm for the maximum flow problem in 1955; the name of the algorithm taking their very surnames: Ford-Fulkerson algorithm.

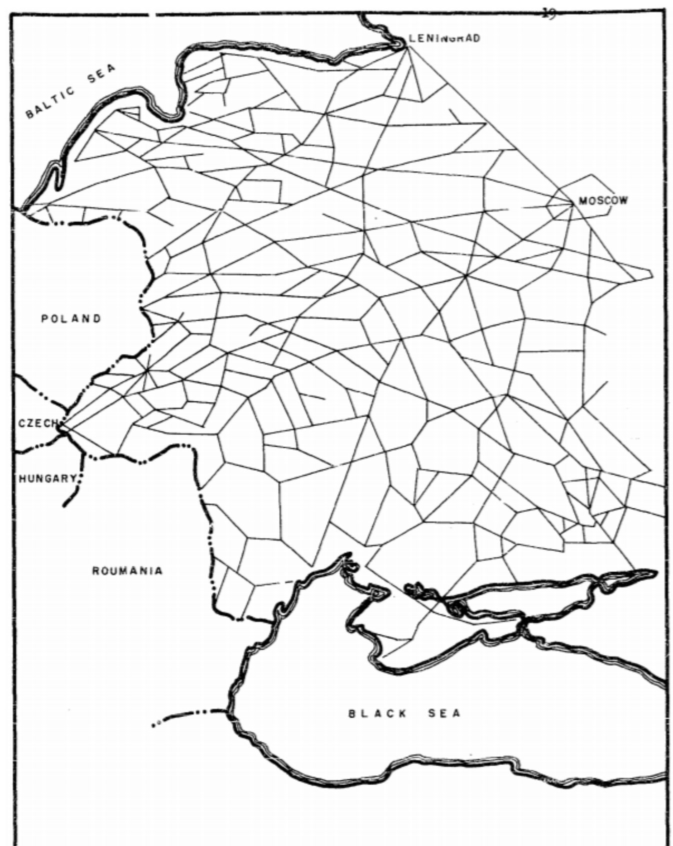


Figure 2. Railway system of western Russia fetched from Ford-Fulkerson's secret documentation
(https://apps.dtic.mil/dtic/tr/fulltext/u2/093458.pdf, accessed on April 24, 2019 23:22 GMT+7)

Ford-Fulkerson algorithm quickly gained attention, mostly because an integral part of solving the problem was not fully specified, leaving a space for others to improvise and develop. In 1970, Dinic's algorithm (or Dinitz's algorithm), a polynomial algorithm for computing the maximum flow in a network was conceived by an Israeli computer scientist Yefim A. Dinitz. Another move was made then in 1972, when Jack Edmonds and Richard Karp independently published their

papers, showing another improvement on the maximum flow problem with optimal utilization of *Breadth First Search* (*BFS)*; thus the name, Edmonds-Karp algorithm.

Since then, many solutions to the problem have come up to public, for instance: the MPM (Malhotra, Pramodh-Kumar, Maheshwari) algorithm, KRT (King, Rao, Tarjan) algorithm, and James B. Orlin's algorithm; each with optimization in different aspects of said problem. In deciding which is better than which in terms of space and time complexity, inspecting individual cases and the network system is essential. Some algorithms are better with less nodes more edges, and vice versa.

II. Basic Theories

A. Directed Graph

A graph G = (V, E) consists of a non-empty set of vertices V and a set of edges E. Each edge has one or a pair of vertices connected with it, called endpoints.

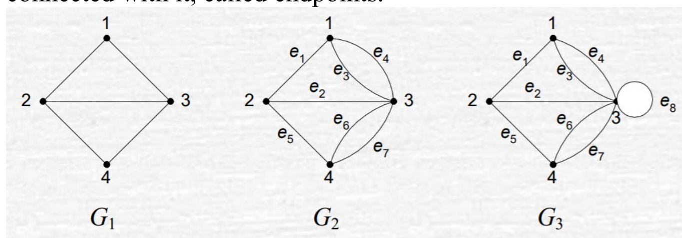


Figure 3. Graphs
(http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2015-2016/Graf%20(2015).pdf, accessed on April 25, 2019 16:35 GMT+7)

For instance, graph G1 in Figure 3 can be considered as G = (V, E) where V = {1, 2, 3, 4} and E = {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}} and G2 can also be considered as G = (V, E) with the same V as G1. However, G2 differs from G1 in terms of its sets of edges E, where E in G2 has 2 sets of parallel or multiple edges, one being e3 & e4, another being e6 & e7. Multiple edges are two different edges connecting the same two vertices; in this case, edges e3 and e4 both connect vertices 1 and 3, while edges e6 and e7 both connect vertices 3 and 4. Edge e8 in G3 is called a loop, as it starts from a vertex (vertex 3) and goes back to the same vertex (back to vertex 3). We can also say that a loop edge connects a vertex to itself.

A graph may have 'direction' in its edges, meaning it has sets of edges made of ordered vertex pair. Such graph is called a directed graph. If it has sets of edges made of unordered vertex pair on the other hand, it is an undirected graph. Graphs G1, G2, G3 in Figure 3 are all undirected graphs as they do not have direction associated with their edges.

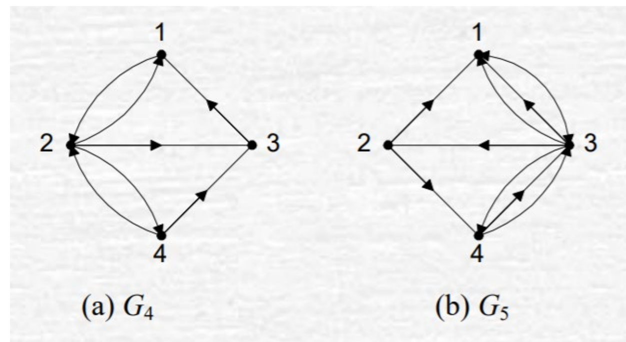


Figure 4. Examples of directed graphs
(http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2015-2016/Graf%20(2015).pdf, accessed on April 25, 2019 16:42 GMT+7)

Graph G4 and G5 from Figure 4 has sets of edges made of ordered pair of vertex, which can be denoted as G = (V, E) where E = (sets of {a, b}), indicating a 'direction' starting from node a to node b, and further implying that {a, b} ≠ {b, a}.

B. Greedy Algorithm

Greedy algorithm is an algorithm paradigm that follows the problem-solving heuristic of making locally optimal choice at each iteration, with the intent of eventually finding a global optimum solution of the problem. In most problems, a greedy algorithm does not usually produce an optimal solution. Nevertheless, a greedy approach may return locally optimal solution approximately close to a global optimum in a much faster time than other algorithms which can ensure globally optimal solution, like brute force (exhaustive search) or divide and conquer.

Greedy algorithm chooses a local optimal solution from a set of choices, fulfilling the objective function of the problem. Objective function is an aspect or value aimed to be maximized. In each iteration, besides from choosing the optimal values and fulfilling the objective function, greedy algorithm also decides if a chosen temporary solution does not violate the feasibility function; for instance, in Integer Knapsack problem, the chosen solution must not exceed the maximum weight for the bag. In some cases, like Kruskal's algorithm and Prim's algorithm for constructing minimum spanning trees of given connected graphs, greedy algorithm effectively always returns optimal solution. However, this effectiveness does not apply to all problems.

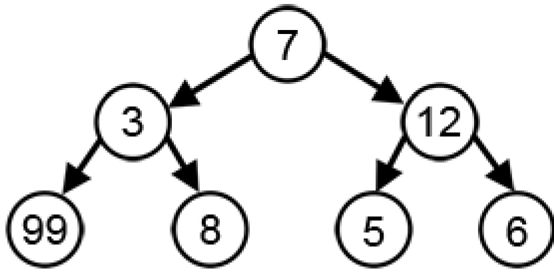


Figure 5. Example of false utilization of greedy algorithm
(https://upload.wikimedia.org/wikipedia/commons/8/8c/Greedy-search-path-example.gif, accessed on April 25, 2019 23:56 GMT+7)

In Figure 4 for instance, with the objective function being to reach the largest sum of numbers from each node by the time it reaches the bottom node. At each step, greedy algorithm will choose what appears to be the optimal solution at the time being, so it will choose node with a value of 12 (right) instead of 3 (left) at the first iteration. Doing so will not reach the global optimum in this problem, which is node with a value of 99 in the left section of the graph. One advantage of using the greedy approach however, despite the result produced being wrong, is the time required to return a solution. Rather than enumerating each potential solution and calculating its total sum of the numbers afterwards (which will take a much longer time as the number of nodes and edges increases), using local optimum is quite acceptable and reasonable, selecting members of solution as it traverses through the graph.

C. Maximum Flow Problem

Maximum flow problem involves computing a feasible flow through a single-source, single-sink flow network that is maximum; with the source and sink each denoted as vertex ‘s’ and vertex ‘t’, respectively. Maximum implies on the most amount of flow to pass through the system; being fed from node ‘s’, passing through the edges along the network system, and eventually ‘coming’ out from terminal node ‘t’. Each edge connecting the nodes in the graph has two values attached; one is the total flow going through said node, and another one is a certain amount of threshold known as the capacity of the edge. Several rules can be directly inferred from the name alone: the amount of flow going through an edge must not exceed its capacity, and the total flow being fed from node ‘s’ to the system must all reach node ‘t’ without any reduction in the amount.

In other version, some algorithms were built for solving the maximum flow problem involving undirected graphs. One of the well-known algorithms is from Sherman and Kelner. The solution to maximum flow problem is equal to the minimum capacity of an s-t cut in the network as stated in max-flow min-cut theorem. Minimum cut is the minimal set of edges required to keep the system working, as well as pursuing for a maximum amount of flow passing through the network.

A formal definition of the maximum flow problem is as follows:

Definition 16.1.1 A flow network is a directed graph $G = (V, E)$ with a source $s \in V$, a sink $t \in V$, and capacities along each edge (described by a function $c : E \to R$ where $c(e)$ is the capacity of edge e). [Prochnow 2009]

Definition 16.1.2 The amount of flow between two vertices is described by a function $f : V \times V \to R$. The flow function f has the following properties:

- Capacity: $f(v,w) \le c(v,w)\ \forall v,w \in V$
- Antisymmetry: $f(v,w) = -f(w,v)\ \forall v,w \in V$
- Conservation: P w∈V $f(v,w) = 0\ \forall v \in V - \{s,t\}$ [Prochnow 2009]

Definition 16.1.3 The total flow $|f|$ of a flow network is the amount of flow going into the sink. Formally, $|f|$ = X v∈V $f(v,t)$. We’re interested in finding the maximum flow, the largest possible $|f|$ for a given graph G. [Prochnow 2009]

Definition 16.1.4 The residual across two vertices $v,w \in V$ is described by function $r : V \times V \to R$ such that $r(v,w) = c(v,w) - f(v,w)$. Thus, the residual $r(v,w)$ represents the amount of potential flow we can still push from v to w. [Prochnow 2009]

Definition 16.1.5 The set of residual edges ER consists of all vertex pairs with positive residuals. Formally, $ER = \{(v,w) \in V \times V \mid r(v,w) > 0\}$. [Prochnow 2009]

III. Approaches on The Maximum Flow Problem: Methods and Algorithms

A. Ford-Fulkerson Method: Flow Augmenting Path Algorithm

The formal algorithm of Ford-Fulkerson method in computing the maximum flow in a network diagram is as follows:

1. Identify the source node and the terminal node. A temporary solution may be set to zero, indicating the flow amount of each edge is 0.
2. Find a flow augmenting path P = (n1, n2, …, np) where n is the nodes in the network along the path.
3. Decide the maximum flow increase in the path specified, that being the capacity of the edge with least capacity along the path. Say we traverse through 5 nodes, we will then have 4 edges, each with capacity of 2, 4, 6, and 8. The maximum flow increase in this path is then 2.
4. Change the flow in each edge, with the amount being the minimum capacity of the edges found in step 3.
5. If a flow decrease needs to happen, mark the flow with a negative number on the edge, indicating that an amount of such flow must be passed through the said edge in order for the solution to be valid. Adjustment happens accordingly.

6. Repeat from step 2 until no other augmenting path can be found, and the algorithm terminates.

The actual method in finding the augmenting paths along the network in step 2 is not specified by Ford and Fulkerson in their papers. For small graphs with no such amount of nodes and edges, only manual observations are required for discovering the augmenting paths. But for larger networks and for computer implementation, a fully defined procedure is required. Ford and Fulkerson, in their documentation, they were using a 'labelling algorithm', where it labels all nodes to which a flow augmenting path from source node 's' can be found. When the terminal node 't' is labeled, the required augmenting path to 't' has been discovered. The algorithm begins with all nodes unlabeled except for the source node 's'. We add an additional label on each node after all avenues for finding paths from the node have been explored (we "check" the node).

When this 'labelling algorithm' terminates and node 't' is labeled, then there is an augmenting path from node 's' to 't', and vice versa. This algorithm is then repeated until no labelling can anymore be made to the nodes. This however, does not guarantee all augmenting paths to be found all the time. Following is an example of the utilization of Ford-Fulkerson method.

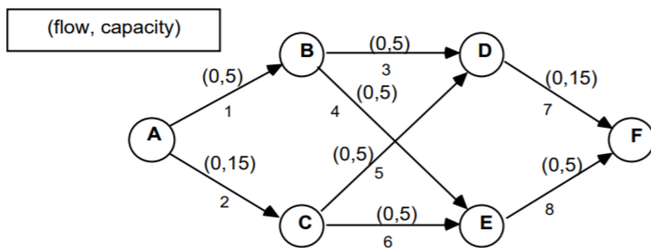


Figure 6. Network flow example, initial condition
(https://www.me.utexas.edu/~jensen/methods/net.pdf/netmaxf.pdf, accessed on April 26, 2019 03:10 GMT+7)

In Figure 6, a network flow diagram is given in a representation of directed graph, with the two numbers in each edge representing the flow and the capacity of the edge, respectively. The source node 's' is node A and the terminal sink 't' is node F. For an initial flow, we feed the system 0 flows, assigning flow number 0 on every edge in the network flow. As a mean of adjusting the solution, the flow inputted from the source node may be increased in an edge when the current flow amount is less than the capacity, and may be decreased if the flow amount violates its capacity.

For the initial iteration in the network flow in Figure 6, there are many augmenting paths we can choose from, as the current flow amount in all edges in the graph is still zero. For instance, we choose path P1 = (A, B, E, F), implying the 'route' travelled passed through nodes A, B, E, and finally node F. As the capacity of the edges along the said path we chose from has the lowest capacity of 5 (in fact, all of them have the same capacity value), the flow may be increased by 5.

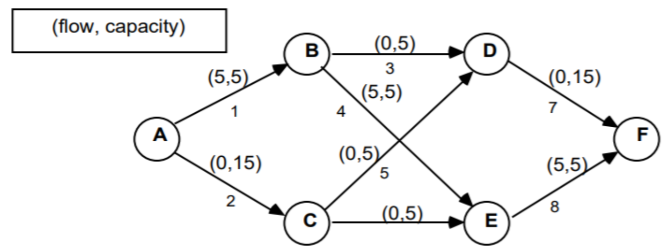


Figure 7. Network flow example, first iteration
(https://www.me.utexas.edu/~jensen/methods/net.pdf/netmaxf.pdf, accessed on April 26, 2019 03:32 GMT+7)

So far, the solution for maximum flow problem in this case is 5. By observation, there is still another obvious augmenting path, P2 = (A, C, D, F). With the utilization of the greedy algorithm, we may find the minimal capacity of the edges along the path P2, which is 5. The flow may then be increased again by 5, updating the solution from 5 to 5+5 = 10.

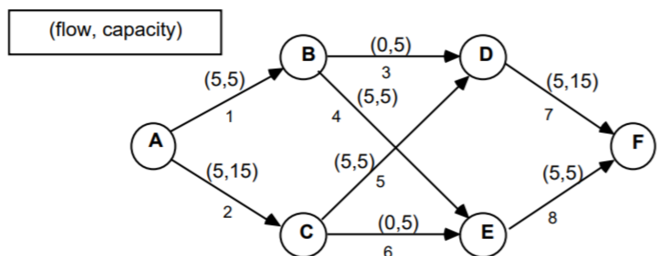


Figure 8. Network flow example, second iteration
(https://www.me.utexas.edu/~jensen/methods/net.pdf/netmaxf.pdf, accessed on April 26, 2019 03:39 GMT+7)

Looking at edge 2 and edge 6, along with edge 1 and edge 3, we discover that there is still one final augmenting path in the network. Say we increase the flow from node A to node C by 5 and deliver the flow to edge 6 afterwards. The flow of amount 5 will then be stuck at node E, since the flow is already equal with the capacity at edge 8 (5 and 5 respectively) and there is no other edge coming out from node E. Looking at edge 3 and edge 4, we can decrease the flow in edge 4 by 5, and adding the decreased value to edge 3, then edge 7 afterwards. The final path updating the network would then be P3 = (A, C, E, B, D, F). A thing to note, when a flow passes through the opposite direction of the directed edge, the flow amount decreases.

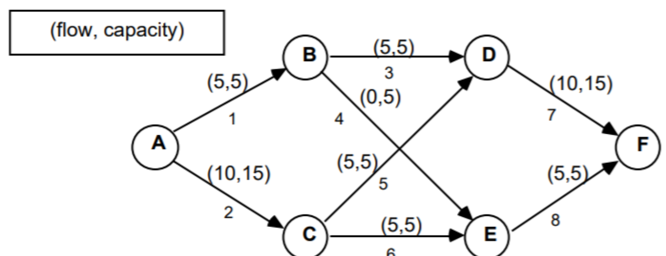


Figure 9. Network flow example, final iteration
(https://www.me.utexas.edu/~jensen/methods/net.pdf/netmaxf.

pdf, accessed on April 26, 2019 04:02 GMT+7)

Looking at the network in Figure 9, it can be deduced that there is no more augmenting path in the network, thus giving us the globally optimal solution of the problem which is 15. To ensure our answer does not violate the constraints and capacities of the edges, we check for the terms of a valid solution:

- The flow being inputted from source node A enters terminal node F without any reduction in amount
- The flow amount in each edge does not go beyond its capacity
- There is no negative amount in the flow amount, indicating that we have made a mistake in mapping the flow and overdone a flow decrease in the network

B. *Edmonds-Karp Algorithm*

Since the 'labelling algorithm' used in Ford-Fulkerson method cannot ensure a solution, a *Breadth First Search (BFS)* implementation on finding the flow augmenting paths in Ford-Fulkerson method is usually used. This very algorithm is called Edmonds-Karp algorithm. Jack Edmonds and Richard Karp independently published an algorithm in $O(VE^2)$ in 1972 in one of their papers *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*, two years after Yefim Dinitz in 1970, where V is the number of nodes/vertices, and E is the number of edges in the network.

Edmonds-Karp algorithm is identical to the Ford-Fulkerson method, except that the process of looking for the flow augmenting paths in the network is specifically defined—implementing Breadth First Search in graph processing. At each iteration, instead of 'labelling' the nodes as it goes, this algorithm applies weight value to the edges. Therefore, in each iteration, instead of checking for the labels in the nodes of the network, we choose the shortest path available which has not been traversed, rather than the path with maximum capacity. By shortest path it means a path with the least amount of edges possible. Although this sounds strange, it is claimed that the algorithm makes at most VE iterations, enabling us to run the whole algorithm including the path finding (the BFS) in $O(VE^2)$. The proof of this theorem is as follows:

"Proof: Let d be the distance from s to t in the current residual graph. We'll prove the result by showing that (a) d never decreases, and (b) every m iterations, d has to increase by at least 1 (which can happen at most n times).

Let's lay out G in levels according to a BFS from s. That is, nodes at level i are distance i away from s, and t is at level d. Now, keeping this layout fixed, let us observe the sequence of paths found and residual graphs produced. Notice that so long as the paths found use only forward edges in this layout, each iteration will cause at least one forward edge to be saturated and removed from the residual graph, and it will add only backward edges. This means first of all that d does not decrease, and secondly that so long as d has not changed (so the paths do use only forward edges), at least one forward edge in this layout gets removed. We can remove forward edges at most m times, so within m iterations either t becomes disconnected (and $d = \infty$) or else we must have used a non-forward edge, implying that d has gone up by 1. We can then re-layout the current residual graph and apply the same argument again, showing that the distance between s and t never decreases, and there can be a gap of size at most m between successive increases. Since the distance between s and t can increase at most n times, this implies that in total we have at most nm iterations." [Blum 2015, Carnegie Mellon University]

Following piece of code is a C++ implementation of Edmonds-Karp algorithm. The data structure used are two 2D-vectors, 'capacity' for storing the capacity of every pair of nodes and 'mat' as a representation of the graph in adjacency matrix. The function 'bfs()' will find the shortest path available from source node 's' to terminal node 't', and will return 0 if no more path is available. Function 'maxFlow()' computes the maximum flow in the given graph, calling 'bfs()' (finding shortest path possible) in each iteration and returns the value when the program terminates.

Code 1.

```
vector<vector<int>> capacity; // storing the
capacity of each edge
vector<vector<int>> mat; // graph representation
in adjacency matrix

// utilizing Breadth First Search (BFS) algorithm
int bfs(int s, int t, vector<int>& prev) {
   fill(prev.begin(), prev.end(), -1); //
initalize previous nodes with -1
   prev[s] = -2;
   queue<pair<int, int>> q;
   q.push({s, INF});

   // while queue is not empty
   while (!q.empty()) {
       int cur = q.front().first;
       int flow = q.front().second;
       q.pop();

       for (int next : mat[cur]) {
           if (prev[next] == -
1 && capacity[cur][next]) {
               prev[next] = cur;
               int new_flow = min(flow,
capacity[cur][next]);
               if (next == t)
                   return new_flow;
               q.push({next, new_flow});
           }
       }
   }

   return 0;
}

// computing maximum flow in the graph
int maxflow(int s, int t) {
    int flow = 0;
    vector<int> prev(n);
```

```
    int new_flow;

    // doing BFS in each iteration, finding the
shortest path available from 's' to 't'
    while (new_flow = bfs(s, t, prev)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = prev[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}
```

C. Implementation and Code Testing in C++

Suppose we have a graph represented in an adjacency matrix, using a 2D vector in C++ as the data structure. Besides the graph, we also have matrix of capacity of the graph's edges, represented too in a 2D-vector. graph[i][j] = 0 indicates that there is no edge from i to j, while graph[i][j] ≠ 0 indicates there exists an edge connecting node i and node j with capacity of capacity[i][j].

Code 2.

```
vector<vector<int>> capacity={{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}
                             };

vector<vector<int>> graph = {{0, 1, 1, 0, 0, 0},
                              {0, 0, 1, 1, 0, 0},
                              {0, 1, 0, 0, 1, 0},
                              {0, 0, 1, 0, 0, 1},
                              {0, 0, 0, 1, 0, 1},
                              {0, 0, 0, 0, 0, 0}
                             };
```

The source node 's' is node 0 as column 0 in the 2D-vector consists of all zeros, while the terminal node 't' is node 5 as row 5 in the 2D-vector consists of all zeros; i.e. no edges are 'going into' node 1 and no edges are 'coming from' node 5, thus the source node and terminal node, respectively.

Code 3.

```
#include <iostream>
using namespace std;

int main(){
vector<vector<int>> capacity={{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}
                             };

vector<vector<int>> graph = {{0, 1, 1, 0, 0, 0},
                              {0, 0, 1, 1, 0, 0},
                              {0, 1, 0, 0, 1, 0},
                              {0, 0, 1, 0, 0, 1},
                              {0, 0, 0, 1, 0, 1},
                              {0, 0, 0, 0, 0, 0}
                             };

    cout << "Maximum flow of the graph:
" << maxflow(0,5) << endl; // as the source node
's' is node 0, and terminal node 't' is node 5

    return 0;
}
```

Passing the source node 's' and terminal node 't' of the graph in Code 2 to function 'maxflow()' in Code 1 in our main program in Code 3 will print to *stdout*:

```
Maximum flow of the graph: 23
```

IV. Conclusion

It began from an unclear 'algorithm' published in 1955, which is basically a method in solving the problem. Next few years, a defined implementation for said method was independently published. Not so long from that, another improvement was found and applied. Since its first formulation until this time, the solution of the maximum flow problem keeps on getting more improvements from the community of computer science, enhancing its algorithm aiming for better efficiency and faster execution time.

The solution of maximum flow problem is not only applicable to the said problem, but also to other similar problems. In fact, it applies to real-life problems, too.

A. Max-flow min-cut theorem

The max-flow min-cut theorem has a similar objective with maximum flow problem, which is to maximize the amount of flow that can pass through a network without violating any constraints. In addition to that, we are to minimize the number of edges that can sustain the system and remove redundant edges from the network.

B. Hall's Theorem and Menger's Theorem

Both of those theorems discuss about graph theory; Hall's theorem (early 1900's) specifically discussing bipartite graphs, while Menger's theorem (1972) discussed about the relationship of the maximum number of edge-disjoint paths from 's' to 't' and minimum number of edges.

C. *Maximizing Traffic and Transportation*

In terms of transportation, the solution of maximum flow problem can help us decide the maximum number of vehicles allowed on a certain sector in the streets as not to cause congestion or other traffic problems.

D. *Maximizing Data and Electicity Transfer*

Data are sent online in packets, though layers of networks. With the solution of maximum flow problem, we are able to send appropriate amount of packets of data through a network as not to cause delay between sessions. The same concept goes for electricity in a wired-system with cables and electrical components.

ACKNOWLEDGMENT

An extremely sincere gratitude towards previous authors whose work was cited in this paper, towards the community of computer science, and towards people's dedication to the development of computer science and technology in general.

REFERENCES

[1] T. E. Harris and F. S. Ross (1955), "Fundamentals of a Method for Evaluating Rail Net Capacities," Research Memorandum, Document Service Centre, US Air Force, Knott Building Dayton 2, Ohio

[2] L. Wang, Y. Chang, and K. Tim Cheng (2009), "Electronic Design Automation: Synthesis, Verification, and Test", revised: Morgan Kaufmann, 2009.

[3] L. R. Ford and D. R. Fulkerson (1956) , "Maximal Flow Through a Network," Canadian Journal of Mathematics. 8: p. 399—404.

[4] J. Edmonds and R. M. Karp (1972), "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". Journal of the ACM. 19: p. 248—264.

[5] R. Munir. (2015). *Graf*. Retrieved from http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2015-2016/Graf%20(2015).pdf

[6] A. Blum, "Network Flow", Carnegie Mellon University lectures. Retrieved from https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1027.pdf

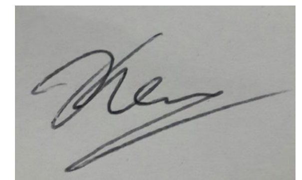
[7] D. A. M. Barrington, "The Edmonds-Karp Heuristic". University of Massachusetts Amherst lectures. Retrieved from https://people.cs.umass.edu/~barring/cs611/lecture/11.pdf

[8] P. Jensen, "Maximum Flow Problem". University of Texas lectures. Retrieved from https://www.me.utexas.edu/~jensen/methods/net.pdf/netmaxf.pdf

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 April 2019



Kevin Angelo 13517086