An Examination of Goldberg and Tarjan's Maximum Flow Algorithm

Samuel McColloch

December 11, 2022

1 Introduction

Systems that allow flow through interconnected nodes exist across a large number of industries. Whether it is fluid flow through pipes, traffic flow on roads, or electronic signals in wires, the behavior of the system can be modeled similarly. In each of these applications it is oftentimes desired to determine the maximum flow possible that the system can handle. This problem is known as the maximum flow problem. The maximum flow problem has been of interest since the mid-20th century. Researchers in the 1950s developed a model for maximum flow known as the Ford-Fulkerson method. Years later Goldberg and Tarjan [4] looked again at the maximum flow problem, implementing a methodology siginificantly different than the classical Ford-Fulkerson method. The Goldberg-Tarjan algorithm incorporated a method known as $Push - Relabel$. The $Push - Relabel$ method produced an overall efficiency of $O(V^2E)$.

2 Motivation and Applications

Since at least World War II, there has been interest in solving the maximum flow problem. The idea had its genesis somewhere between World War II and the 1950s. Early conceptualization came forth from military strategists looking to cut a minimum number of routes to cause maximum disruption to the enemy. In the 1950s, Ford and Fulkerson approached the maximum flow problem with the idea of railways. The authors imagined a system of many rail lines connecting through many cities [3]. In computer science, this network of nodes and connections that allow flow are often viewed as a weighted directed graph data structure. The nodes are the vertices, the edges are the connections, and the flow capacity is the weight of the edge. A number of solutions to the max flow problem approach it as a weighted directed graph. [2]

The solution presented by Ford and Fulkerson involved the minimum cut theorem and using shortest paths. In the 1970s, Alexander Karzanov developed the idea of $preflow$ which allows a single node to be modified instead of the entire path of the system [6]. Other researchers looked at the maximum flow problem and created methodologies that converted Ford and Fulkerson's implementation into polynomial time.

Goldberg and Tarjan took Karzanov's concept of $preflow$ and developed a new methodology for calculating the maximum flow. The Goldberg-Tarjan algorithm uses two key functionalities, push and relabel. Adding these two functionalities allowed for greater

performance and an overall faster algorithm than some of the previous implementations [5].

The solution to the maximum flow problem has applications in many sectors. Any system with flow from a source to a destination has a use. It has applications to oil and gas transport, water pipelines, networking, transportation, digital imaging, electricity and even aligning job skills with employers[1]. Increasing the efficiency of the process to find the maximum flow in these systems is of utmost importance. The improvements to the runtime by Goldberg and Tarjan have wide ranging uses in any of the previously mentioned industries.

3 Description and Psuedocode

The algorithm works in stages, focusing on an individual vertex at a time. The algorithm begins with selecting a source and a sink based on user input. The source is the origin point of whatever is flowing through the system while the sink is the end point. Once the source and sink have been identified some initialization of variables occurs. The graph is made of vertices and edges. In order to implement the algorithm the edges and vertices need additional parameters linked to them. In some sources, the edges are referred to as "arcs". An arc is simply an edge to the graph that has an additional parameter known as preflow linked to it. [5] While the algorithm has practical applications in many fields, for the purposes of explanation the system will be viewed as pipes with fluid flow.

Each of the arcs can be viewed as a pipe. The pipe has a certain capacity or flow rate and this is the weight of the graph. The author's often refer to it as capacity. This is the maximum amount of fluid that can ever be in the pipe. Next, the concept of preflow is introduced. The preflow is a measure of how much fluid is currently in the pipe. It starts at zero and as the algorithm progresses, fluid is forced from one section to the next. The value is dynamic and is updated repeatedly during the algorithm's execution.

Each of the vertices in the graph also have values associated with them. Each vertex has a height (sometimes called distance) [2] and a value for excess flow. Excess flow can be thought of as the amount of fluid sitting at a node that needs to be dispersed. The goal is to get as much fluid from the source to the sink and have no excess fluid on any node at the end.

Once a source and sink are chosen the algorithm begins. The arcs and vertices are initialized based on their proximity to the source. Any outflow arc immediately connected to the source has its preflow initialized to the value of its capacity. The vertices that are also directly connected to the source have their excess flow initialized to the capacity of the arc that connects to them from the source. The height parameter of all the vertices except the source are initialized to zero. The source vertex has its height initialized to the number of vertices in the system. The height of each vertex tells the algorithm which direction fluid flow is allowed. As in real life, fluid will only flow from high ground to low. So in order to have fluid move from one vertex to the next the height of the vertex must be one unit higher than the one it flows to.

The vertices that are adjacent to the source and have been initialized are now considered to be active. A vertex remains active as long as it has excess flow greater than zero. The algorithm has two basic operation that it continually conducts until the only nodes with excess fluid are the source and sink. Or in other words, there are no more active vertices. The first and simpler operation is *Relabel*. *Relabel* works as an iterator that adjusts the

height of a vertex to allow it to flow to a vertex with a lower height. The operation is only in one direction and the height of a vertex is never decreased once it has been increased. [4]

The other main operation the algorithm uses is $Push$. Before a push operation can be performed a check is done to see if there is an available vertex to push to. An available vertex that can be pushed to must be a direct neighbor to the active vertex with a height that is one unit less than the active vertex. The arc that connects the two vertices must also have a residual capacity that is greater than zero. The residual capacity of the pipe is simply the differential of the capacity and its current preflow. So, in other words, if there is room in the pipe for fluid and the vertex it is connected to is lower than the active one, it can be pushed to. If no suitable vertex exists due to inappropriate height differentials, the *relabel* function is called until a neighbor can be found to push to. Additionally if all neighbors are at full capacity then the excess flow must be directed back to the source. The fluid flows back upstream until it reaches the source. Any intermediate vertex will have to have its height adjusted as needed to allow flow to upstream vertices.

Once a suitable vertex to push to has been determined the $Push$ operation is done. $Push$ takes a vertex pair (u, w) and calculates how much flow can be sent through the arc to the neighbor vertex. This is done by a comparison between the excess flow at the active vertex and the current residual capacity of the arc. The minimum value between these two is chosen as a Δ_f. This delta is now moved from the active vertex through the arc, to the neighbor vertex. The action removes delta from the active vertex, adds it to the preflow of the arc, and adds it to the excess of the neighbor vertex. The neighbor that received the flow becomes an active vertex. If the current active vertex still has excess flow, it remains active and another path for the flow will be sought. If the excess flow is zero, the vertex becomes inactive. If there is no downstream path for the fluid to flow due to each arc reaching capacity, then the excess flow must be sent back to the source.

The algorithm keeps looping as long as there are active vertices. Once all active vertices have had their excess flow reduced to zero the algorithm terminates. The value of the excess flow at the sink is returned and this is the maximum flow of the system. The psuedocode for the maximum flow algorithm and each of its helper functions can be seen below. An implementation of the algorithm was created and will be discussed in detail later on. The psuedocode presented here largely matches the implementation. The psuedocode from the original paper had some ambiguity in its presentation and this code is presented to help clarify the process.

Algorithm 1

MaxFlow: Finds the maximum flow in a weighted directed graph using the Push-Relabel method described by Goldberg and Tarjan

INPUTS:

n: the number of vertices in the graph

E: a list of weighted edges of the form (u, v, c) where c is the capacity

s: the source vertex, selected as the origin of the flow

t: the sink vertex, selected as where flow terminates

OUTPUT:

MaxFlow: an integer that represents the maximum flow the system can handle

```
1: function MaxFlow(n, E, s, t)
2:     A = InitializeArcs(E, s) ▷ dict key: edge, value: flow, capacity
3:     B ▷ empty dictionary for backflow
4:     V = InitializeVertices(n, s) ▷ dict key: n, value: height, excess flow
5:     N = InitializeNeighbors(E) ▷ dict of neighbors with directionality
6:     Q_active = N[s] ▷ active vertices, behaves as a queue, v_active is front of queue
7:     V[Q_active] = A[s, v]_capacity ▷ active vertices get the capacity of arcs that connect to
       source
8:     while |Q_active| > 0 do
9:         w = AbleToPush(v_active, V, A, N) ▷ vertex of a neighbor that can be pushed
10:        if suitable neighbor is found then
11:            Push(v_active, w, V, A, B)
12:        else
13:            Relabel(v_active, V)
14:        if V[v_active]_excessflow = 0 then
15:            Q_active.Dequeue() ▷ removes first element from queue
16:        if v_push has flow and is not the source or sink then
17:            Q_active.Enqueue(w)
       return V[t]_excessflow ▷ returns the excess flow at the sink, which is max flow
```

Algorithm 2
AbleToPush
INPUTS: v, V, A, N
OUTPUT: neighbor to push and direction of flow

```
1: function AbleToPush(v, V, A, N)
2:     W = N[v] ▷ get all neighbors to active vertex
3:     n_full = 0 ▷ number of downstream neighbors at full capacity
4:     n_down = 0 ▷ number of downstream neighbors
5:     for w = 0 to |W| do
6:         if W[w]_fd = upstream then ▷ checks flow direction of current neighbor
7:             Break
8:         rc = A[v, W[w]]_capacity − A[v, W[w]]_preflow ▷ residual capacity of the arc
9:         if rc > 0 and v_height = W[w]_height + 1 then
10:            Return W[w]
11:        if rc == 0 then
12:            n_full = n_full + 1
           n_down = n_down + 1
13:    if n_full = n_down and v_height = W[w]_height + 1 then
14:        Return W[w]
15:    Return (−1, −1) ▷ Indicates no suitable neighbor was found
```

Algorithm 3
Relabel: Increments the height of a vertex by one.
INPUTS:
v: current vertex
V: dictionary that contains all the vertices

```
1: function Relabel(v, V)
2:     V[v]_height = V[v]_height + 1
```

Algorithm 4
Push: Updates the preflow values in arcs and excess flow in vertices. Pushes the flow from an active vertex to a neighbor vertex
INPUTS:
v: active vertex
w: vertex to push to
V: all vertices
A: all arcs
B: backflow arcs

```
1: function PUSH(v, w, V, A, B)
2:     if w is downstream then ▷ flow will be pushed downstream to a neighbor
3:         rc = A[v, w]_capacity − A[v, w]_preflow ▷ residual capacity
4:         Δ_f = min(v_excessflow, rc)
5:         A[v, w]_preflow = A[v, w]_preflow + Δ_f
6:     else▷ When downstream is full, the excess gets pushed back toward source
7:         Δ_f = v_excessflow
8:         B[v, w] = Δ_f
9:         if w = source then
10:            subtract backflow from arcs
11:    v_excessflow = v_excessflow − Δ_f
12:    w_excessflow = w_excessflow + Δ_f
```

4 Implementation

An implementation of the algorithm was created using Python. The implementation followed the primary methods as outlined in the original paper but some modifications were made due to ambiguity in the original pseudocode. The Sequential implementation as outlined by Goldberg and Tarjan uses a queue mechanism to maintain the active vertices. This FIFO queue was used in the implementation.[4] The original paper offers very little detail on the code around how it is determined if you can push to a vertex. They give the condition that the residual capacity must be greater than zero and the height must be one unit higher. However, when implementing the algorithm it became apparent that there needs to be some way to tell the push function if the flow should be directed downstream or upstream.

In order to achieve this a flag was used that kept track of how many downstream neighbors had reached full capacity. Downstream neighbors are indicated by a 0 and upstream neighbors indicated by a 1. Once all downstream neighbors are full, the *able_to_push* function returns the neighbor with the flag to the max_flow function and that tells the direction of flow when *push* is called.

When the flow is pushed back to the source, the amount that passes through each arc also needs to be tracked so that it can be removed. A dictionary called $backflow\_arcs$ that stored the flow that needed to be removed from each arc was implemented. When the flow reached the source vertex another function called $remove\_backflow$ was implemented that adjusted the flow rates in the arcs accordingly.

The rest of the implementation was fairly similar to the psuedocode, a FIFO queue was used to maintain the active vertices and the push and relabel functions operated

```
Maximum Flow: 23

Flow in Arcs at End (Preflow / Capacity):

Arc (0,1): 11 / 11
Arc (0,2): 12 / 12
Arc (2,1): 1 / 1
Arc (1,3): 12 / 12
Arc (2,4): 11 / 11
Arc (4,3): 7 / 7
Arc (3,5): 19 / 19
Arc (4,5): 4 / 4

Excess Flow and Height of Vertex at End (Excess Flow | Height):

Source:      0 | 6
Vertex 1:    0 | 1
Vertex 2:    0 | 2
Vertex 3:    0 | 1
Vertex 4:    0 | 2
Sink:       23 | 0
```

appoutput

Figure 1: Example output of the maximum flow algorithm application

in the manner described in the previous section. Several test cases were created for the *max_flow* function and it was able to match expected output. The program also has an console application to run the algorithm and produce a meaningful output.

maxflow.py ▷ *Contains the implementation of the max flow algorithm*
maxflow_application.py ▷ *a driver program that takes a text file as argument and produces an output with the state of the system at the end*

The input file is a text file with the form:

6 ▷ *number of vertices*
0 ▷ *source*
5 ▷ *sink*
0,11,11 ▷ *Edge: start, end, weight*
0,2,12 ▷ *edges are assumed to flow from start to end*
⋮
4,5,4

On the command line an example run would be:

python3.10 -m maxflow_application.py input.txt

The program will produce an output that shows the state of the system at the end of execution. The maximum flow is reported at the top. It also shows the amount of preflow in each arc compared with the total capacity of the arc. It also shows the excess flow in all vertices including source and sink. The excess flow of all vertices except the source and sink should be zero. The final height of each vertex is also reported. An example output can be seen in Fig. 1.

5 Correctness

The correctness of the algorithm can be shown through a series of lemmas and associated proofs.

Lemma 1: If a vertex has excess flow it is said to be active. For any active vertex there is a path that connects it to to the source.

At the beginning of the algorithm flow is dispersed from the source to the adjacent arcs and vertices. As the algorithm executes the flow is pushed along, therefore a vertex could never have excess flow if there is no path from the source. [4] If we let $e(w)$ be the excess flow at a vertex and $f(u, w)$ be the flow from a vertex, with S being the set of vertices connected to v and $\overline{S}$ be vertices that are not reachable by v it can be shown that:

$$\sum_{w \in S} e(w) = \sum_{u \in \overline{S}, w \in S} f(u, w) \leq 0$$

So for all vertices in S $e(v) = 0$.

Lemma 2: The sink is not directly reachable by the source

If each vertex v is between the source and sink, then there is a length $l < n$ and a pair of vertices within the list of arcs. It can be assumed that the height $h(v_i) \leq h(v_{i+1})$ for all $0 \leq i < l$. The source height is always initialized to the number of vertices which is n while the sink height is initialized to 0. This gives that $h(s) \leq h(t) + l < n$ and $h(t) = 0$. Since the sink's height $h(s) = 0$ then this gives a contradiction because $h(s) = n$. [4].

Lemma 3: The height of a vertex will never decrease and the height is bounded by: $h(v) \leq 2n - 1$

The height of a vertex will obviously never decrease as it is determined strictly by a function that is additive. The path from the active vertex to the source has a worst case possibility of having to pass through each vertex to reach the source, this gives a length of n - 1. The initial height of the source is equal to n. Or $h(s) = n$. If a vertex had to traverse the full path and overtake the source's height that gives a maximum value of $n + (n - 1)$ or $2n - 1$. [4]

Lemma 4: The maximum number of relabel calls per vertex will not be $> 2n - 1$.

In Lemma 4 it was shown how the maximum bounding on a vertex is $2n - 1$. Intuitively, this also means that the maximum number of times any vertex could be relabeled is $2n - 1$. [4]

Lemma 5: $2mn$ *is the maximum number of push operations*

Push from one vertex to the next is determined by the capacity and the height. Therefore if a vertex v pushed to its neighbor w this means the height at v is one unit above w. So in order for the flow to be pushed back to v, w would have to increment by two units. Lemma 4 and 5 showed the maximum possible height a vertex can reach. Therefore, it

would mean $(2n-1)m < 2nm$, where m is the number of edges. [4]

6 Runtime

The sequential algorithm runs in $\mathcal{O}(V^2E)$. The algorithm will move from active vertex to active vertex which will take linear time. Once an active vertex is selected the neighbors adjacent to it will be examined, a process that also takes linear time. This gives an expected time of $\mathcal{O}(V^2)$ for the vertices comparisons alone. Flow is pushed to chosen edges but thanks to a dictionary this search is in constant time $\mathcal{O}(1)$. Each edge will most likely be populated with flow though and this will add another linear operation of $\mathcal{O}(E)$. Overall complexity is:

$$\mathcal{O}(V^2) + \mathcal{O}(1) + \mathcal{O}(E) = \mathcal{O}(V^2E)$$

There are more complex implementations of the algorithm that have incorporated dynamic trees that have been able to achieve $\mathcal{O}(nmlog(n^2/m))$ time.

7 Conclusion

The maximum flow problem has been well known with solutions going back decades. The work of Ford and Fulkerson help lay the groundwork for solving the maximum flow problem. Through the years numerous other implementations have built upon their initial solution. Goldberg and Tarjan approached the problem quite differently and were able to achieve a more efficient solution with polynomial time complexity. The implementation created was able to successfully follow the sequential methodology of the algorithm. The relative ease of implementation shows how this solution to the maximum flow problem has wide spread applications in in any system that deals with flow.

References

ahuja [1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, New Jersey: Prentice-Hall, 1993.

cormen [2] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. London, England: The MIT Press, 2009.

fordfulk [3] L.R. Ford and D.R. Fulkerson. "Maximal Flow Through Network". In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. DOI: 10.4153/CJM-1956-045-5.

lbergtarjan [4] Andrew V. Goldberg and Robert E. Tarjan. "A New Approach to the Maximum-Flow Problem". In: *Journal of the Association for Computing Machinery* 35.4 (1988), pp. 921–940. DOI: 10.1145/48014.61051.

ergtarjan2 [5] Andrew V. Goldberg and Robert E. Tarjan. "Efficient Maximum Flow Algorithms". In: *Communications of the ACM* 57.8 (2014). DOI: 10.1145/2628036.

karz [6] Alexander Karzanov. “Determining the Maximal Flow in a Network By the Method of Preflows”. In: *Soviet Mathematical Dokladi* 15 (1974), pp. 434–437.