University of Pennsylvania

Final Independent Study Project

Network Flow Algorithms and Their Applications

Ketan Mandava

supervised by
Dr. Rakesh Vohra

June 2022
Github Repository: `https://github.com/ketanm1999/network-flows-ketanm`

Contents

List of Figures

List of Listings

1 Introduction

1.1 The Network Flow Problem

Give an example of the core network flow problem

Figure 1: Simple Network Flow Problem

2 The Shortest Path Problem

Let us say that an international retail company with only one manufacturing site based in China is attempting to place storage locations throughout Asia to reduce the travel-time of one unit of their supply to reach London, England. The approach to this problem might seem relatively simple if the retail company only has one or two potential storage locations, but what if they had five potential options? Ten? What if the retailer has the potential to reduce travel-time by sending parcels through multiple storage locations before reaching London instead of sending them direct? Such a problem has the potential to become complicated as you add more potential storage locations to the system. This problem is exactly like a shortest path problem.

The shortest path problem is likely most integral network flow problem that will be addressed in this paper. While it is not very complex, it plays a pivotal role in defining network flows in various contexts and has numerous real world applications. In addition to the scenario addressed above, potential real world situations where such a network exists are distributed networks such as AWS, transportation networks such as Google Maps, airline planning (earliest arrival time at a specific airport), and many others.

The shortest path problem at its core can be reduced to having two components: a set of directional arcs or edges, E; and a set of nodes through which these arcs pass, V. Each arc in our network has a distance or cost associated with it, and two of our nodes will represent the source of our network (denoted s) and the destination (denoted t) of our network. The goal of such a problem is to find the shortest path from the source node to the terminal node through the arcs of the network. A path can be defined as a sequence of arcs where each arc begins with the destination node of its predecessor and ends with the source node of successor. For example, lets say we have Nodes $V = \{1, 2, 3, 4\}$ and Arcs $E = \{(1, 2), (2, 3), (3, 4), (4, 2)\}$, where each arc is a pair of nodes with the second node representing the destination of the arc. In this network, some potential paths could be $\{(1, 2), (2, 3), (3, 4)\}$, $\{((2, 3), (3, 4), (4, 2)\}$, or $\{(1, 2), (2, 3)\}$ because they all satisfy the aforementioned criteria.

The shortest path problem can be seen as a simplification of the much more complicated minimum cost/maximum flow problem that will be discussed later in this paper, and is a pretty good introduction to the idea of costs in a network. Something you may have noticed is the interchangeability of cost and distance

in my explanation of these problems. This is because the shortest distance problem can take on both a minimum cost or minimum distance form.

Now that we have a better understanding of what these problems are, we will discuss some potential ways of solving problems like these. In this section we will be discussing two algorithms for solving shortest path problems, those being Dijkstra's Algorithm and the Bellman-Ford Algorithm. Both algorithms can achieve the goal of finding the shortest path in a network, but they are by no means the same.

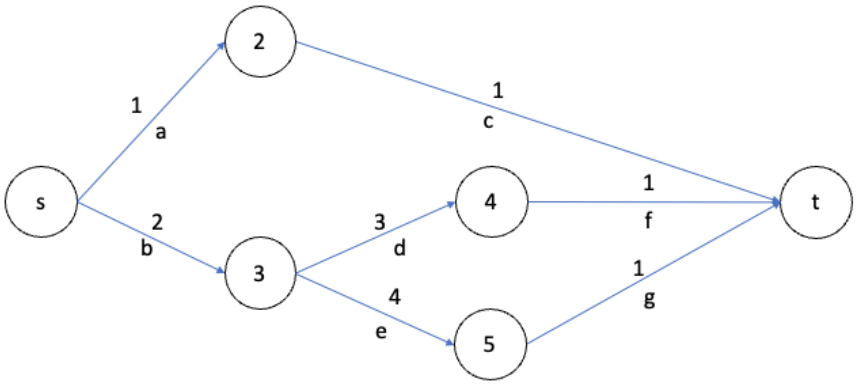

Figure 2: Shortest Path - Example Problem
Here, each edge has a name (alphabetical character) and a cost/distance (numerical character), and we have 6 nodes, $\{s, 2, 3, 4, 5, t\}$.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm is likely the most well known shortest path algorithm out there. Found in 1956 by Edsger Dijkstra, this algorithm has been taken on many forms, with its original form being finding the shortest path from a source node to a destination node in a graph as described in the previous explanation. This algorithm is very useful and not extremely complex, but can be somewhat restrictive when it comes to what graphs it can solve.

The algorithm has a few rules for what it is capable of handling. First, all costs in the graph must be non-negative - if negative costs exist in the graph, Dijkstra's algorithm has the potential to return an incorrect shortest path. Second, there must not exist negative cycles in the graph - Dijkstra's algorithm will enter an infinite loop if any negative cycles exist, the reason for which will be explored below.

The core of the algorithm relies on the construction of a predicted distance label on a given node d_i, representing the algorithms current prediction for the shortest distance from the source node in the graph to the given node. Initially, all nodes have $d_i = +\infty$, where the algorithm is effectively placing an upper

bound on the correct shortest path to each node. The only exception to this rule is that the initial value of $d_s = 0$ because the distance from the source to itself is 0. The value of d_i for all other nodes is only updated if the algorithm finds a path to i, lets say from another node j, where $d_j + c_{ji} < d_i$, or the current predicted minimum distance to j from s plus the additional distance along the arc between j and i is **less than** the current predicted distance to i.

Additionally, each node can either be marked or unmarked. A node is marked when the algorithm is certain that its d_i equals the true shortest path to it from the source, and is unmarked otherwise. Finally, each node has a pointer variable p_i which points to the previous node in its shortest distance path from s. Because the algorithm does not initially know which node will be the predecessor to a given node, each node has $p_i =$ **null** initially.

Now that we have the groundwork for the algorithm, how do we start? The algorithm can be broken up into several simple steps:

1. Searching the node set for the node, i, that is unmarked and has minimal d_i. The algorithm will intuitively begin with s as it is the only node with $d_i < +\infty$.

2. From here, it searches the arc set for arcs whose source is the selected node. It then tests the destination node, j, of each arc to see if $d_i + c_{ij} < d_j$. *****

3. If and **only** if this inequality holds true, the destination node is updated to $d_j = d_i + c_{ij}$ and $p_j = i$.

4. Mark i.

5. Return to step 2 until all nodes are marked.

6. When all nodes marked, pull d_t, which will be our shortest distance value.

That's it! This algorithm is relatively simple but is comprehensive. The algorithm operates on $O(n^2)$ time complexity, where $n = |V|$ or the total number of nodes in the graph. One important thing to note is the fact that Dijkstra's algorithm only returns the integer value of the shortest distance, not the shortest path itself. The shortest path can be found by backtracking through the pointer variables from the terminal node to the source node. Now, lets explore the negative cost and negative cycle issues with this particular algorithm. Let's begin with the negative cycle issue first.

A cycle can be defined as a path (recall the definition of a path as a sequence of arcs where each arc begins with the destination node of its predecessor and ends with the source node of successor) through which if you select a node and continue through the path from this node, you will come across the same node. From this, a negative cycle can be defined as a cycle in which the sum of all costs is negative. Mathematically, if we define a set C as the arcs within a the given cycle, then if $\sum_{i,j \epsilon C} c_{ij} < 0$, the cycle is a negative cycle. Recall that we update a given node i's d_i if and only if $d_j + c_{ji} < d_i$. Within a negative cycle, each successive iteration will decrease the d_i of each node in the cycle by

$\sum_{i,j \epsilon C} c_{ij}$ and, thus, $d_j + c_{ji} < d_i$ will always hold true, resulting in an infinite cycle.

Now that we understand why a negative cycle will cause Dijkstra's algorithm to fail, let us think about what happens if we have a negative cost edge but no negative cycle. Let us say we have the graph in figure 3 with the arc set $E = \{(s,1),(1,2),(s,2),(2,t)\}$ with the node set $V = \{s,1,2,t\}$. We can see that there exists no cycles in the graph, so lets run our algorithm and see what happens.

1. We begin from node s, whose $d_s = 0$, and search the arc set for arcs whose source is the selected node and destination is unmarked. We find $(s,1)$ and $(s,2)$.
2. We test nodes 1 and 2 with $d_i + c_{ij} < d_j$, finding both to be true ($d_s + c_{s1} = 4 < d_1 = +\infty$, $d_s + c_{s2} = 2 < d_2 = +\infty$
3. We update nodes 1 and 2: $d_1 = d_s + c_{s1} = 4$, $d_2 = d_s + c_{s2} = 2$, $p_1 = s$, $p_2 = s$
4. We mark s.
5. We then move to node 2 because it has minimum distance of all unmarked nodes. We perform the same search as in step 1 and find only $(2,t)$
6. We perform step 2 on t and find it true ($d_2 + c_{2t} = 3 < d_t = +\infty$)
7. We update t. $d_t = d_2 + c_{2t} = 3$, $p_t = 2$.
8. We mark 2.
9. We now select t as it has minimum d_i of unmarked nodes and perform the same search as in step 1, but yield no arcs. So, we mark t.
10. We now select 1 as it is the last unmarked node and perform step one, but yield no arcs so mark 1.

    END

So, Dijkstra's algorithm will return a shortest distance value of $d_t = 3$ on this graph. One should intuitively notice that this value is incorrect, as the path $\{(s,1),(1,2),(2,t)\}$ would yield a shortest distance value of 2, which is less than the value Dijkstra returned. This issue arises because Dijkstra is extremely preemptive when in comes to marking nodes, and when a node is marked the algorithm effectively locks its distance value as true. So, when node 2 is selected in step 5, the algorithm is already on track to fail.

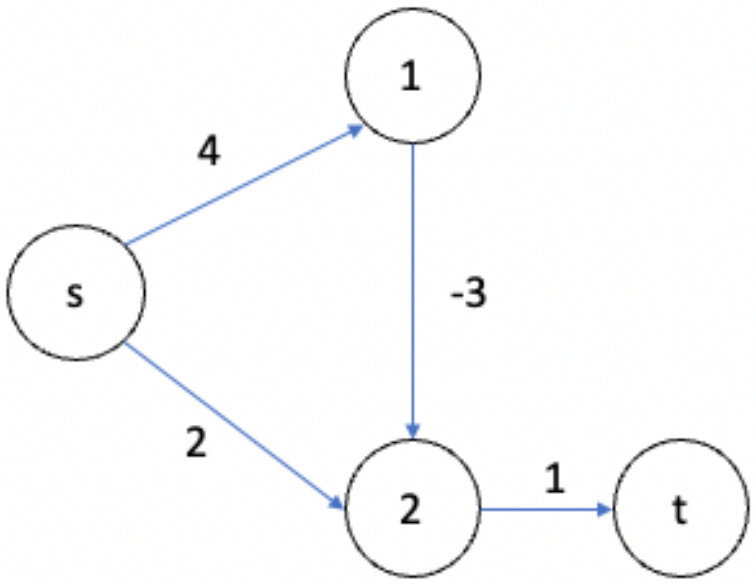

Figure 3: Dijkstra - Incorrect Path With Negative Cost Edge

Now that we understand what Dijkstra is capable of, lets move on to Bellman-Ford, which has some capabilities that Dijkstra does not.

2.2 The Bellman-Ford Algorithm

The Bellman-Ford Algorithm is extremely similar to Dijkstra's Algorithm, but differs in its recursive nature. The setup to the algorithm is almost identical to that of Dijkstra's because Bellman-Ford also relies on a distance prediction variable d_i as well as a pointer variable p_i on each node. However, Bellman-Ford does not utilize a marked Boolean value and rather relies on a theorem that we can address after exploring the algorithm itself. The algorithm is as follows.

1. Set $d_i = +\infty$ and $p_i =$ **null** for all nodes (initial distance to each node is unknown and initial predecessor is unknown). Set $d_s = 0$ (explanation described in 2.1).

2. Relax all d_i $|V| - 1$ times through the following:

For each arc (i, j):

if $d_i + c_{ij} < d_j$, then $d_j = d_i + c_{ij}$ and $p_j = i$

3. Repeat search for one more iteration: For each arc (i, j)

   if $d_i + c_{ij} < d_j$, then END (there exists a negative cycle and there is no shortest path)

4. Return d_t as the shortest distance value.

This appears to be much more confusing than Dijkstra's algorithm at first glance. It is helpful to break down why each step is performed, beginning with step 2 which is where this algorithm first differs from Dijkstra.

Step 2 performs $|V| - 1$ iterations of $d_i + c_{ij} < d_j$ on each arc in the graph. This is where the aforementioned theorem comes into play. According to Bellman-Ford, completing $|V| - 1$ iterations of relaxation on each arc will provide the true shortest path in the graph. How can this be true? It becomes extremely clear why this is possible by proving the Bellman-Ford Correctness Theorem.

Theorem 2.1 (Bellman-Ford Correctness)**.** *If no negative cycles exist in the graph $G = (V, E)$, after completing Bellman-Ford, then all d_i is true and the returned shortest distance, d_t is true.*

Proof. If no negative cost cycles exist in G, then a shortest path must exist.

For any node i, let d_i^k be the length of the shortest path from s to i that utilizes no more than k edges. If such a path does not exist, then $d_i^k = +\infty$. We can prove $d_{i,TRUE} \leq d_i^k$.

Let P be some simple path (a path with no repeated nodes) from s to t. P must be constructed of **at most** $|V|$ nodes. Each node in a path is connected by one Arc. Therefore, P must be constructed with no more than $|V| - 1$. □

Because of the above, it becomes clear that after $|V| - 1$ iterations, Bellman-Ford finds the true shortest path! What does step 3 do then? Recall that the theorem above only holds true if there are no negative cycles in the graph. But what if there are negative cycles? This is where step 3 comes into play. Bellman-Ford has a built in fail-safe for graphs with negative cycles! While this will not be necessary for our purposes here, it will become extremely important in part 5 of this paper (hint, hint). It is important to note that Bellman-Ford is additionally capable of handling negative cost arcs while Dijkstra can't due to it's redundancy. While this can be seen as a benefit, it can be viewed as a detriment if your primary concern is run-time - Bellman-Ford has a time complexity of $O(nm)$, where $n = |V|$ and $m = |E|$. This is substantially larger run-time than Dijkstra, which can lead to issues with larger data-sets.

3 The Maximum Flow Problem

The maximum flow problem is another extremely important network flow problem that constructs the backbone of almost all of the algorithms discussed. Just

as was the case with the shortest path problem, the maximum flow problem can be viewed as the simplest version of some other more complex network flows and can serve as a very easy introduction to the idea of capacity.

The core of the maximum flow problem is constructed of the same two components we dealt with previously: a set of directional arcs or edges, E; and a set of nodes through which these arcs pass, V. We still have two of our nodes representing the source of our network (denoted s) and the destination (denoted t) of our network, but whereas in the shortest path problem our arcs had a distance or cost parameter, here our arcs have a **capacity** parameter, u_{ij}. We additionally have the variable x_{ij} on each arc, representing the flow on that specific arc. With this, we also have flow constraints that were not mentioned in the previous section. The first is the flow conservation constraint, that being the the sum of all inflows to a node must be equal to the sum of all outflows from that node. The second is the capacity constraint, that being that the flow along an arc must not exceed that arcs capacity and must be non-negative. Mathematically, these constraints can be represented as the following:

$$\sum_{(i,j)\epsilon E} x_{ij} - \sum_{(j,i)\epsilon E} x_{ji} = 0 \; \forall \; i, j \epsilon V \tag{1}$$

$$0 \leq x_{ij} \leq u_{ij} \; \forall \; i, j \epsilon V \tag{2}$$

The goal of this problem is to push the maximum amount flow through the system from s to t without exceeding the capacity along each arc.

Before we explore an algorithm for solving such problems, it can be beneficial to first explore an example of such a problem. In figure 4, we have an example maximum flow problem graph, where each arc has a capacity (numerical value) and a name (alphabetical value), and each node has a name (alphanumerical value). It is already extremely difficult to understand what exactly could be the potential solution for maximum flow! It may be helpful for us to first explore the idea of a cut before exploring maximum flow in this graph.

Definition 1. *An $s-t$ cut is a set of arcs that, when removed from the graph, disconnect s from t. The capacity of an $s-t$ cut is equal to the the sum of the capacity along each arc in the cut.*

Now that we know what an $s-t$ cut is, lets try to find one in figure 4. One example of $s-t$ cut in this graph could be the cut set $C(S,T) = \{(s,2),(3,5),(3,4)\}$. With these arcs removed from the graph, it becomes impossible to connect s to t through the graph. In other words, the graph is partitioned into two parts, with one containing s (we can call this A) and one containing t (we can call this B).

A cut becomes significant to our discussion of maximum flow when we consider a **minimum cut**. A minimum cut is a cut in the graph with minimum capacity. Our particular graph has only one minimum cut, that being the cut $C(S,T) = \{(2,t),(4,t),(3,5)\}$ with capacity 23. What significance does this value have? Well, lets consider what capacity in our graph exactly means. Each arc's capacity dictates the **maximum** amount of flow that can be exist on that

edge. If we look at the arc $(2, t)$, we can see that there is only one potential path from s to t through this arc, that being $\{(s, 2), (2, t)\}$. We can therefore refer to $(2, t)$ as the bottleneck arc in that path, where only 5 units of flow can be sent along that path. If we now look at $(3, 5)$, we can see that it is also the bottleneck arc of its path, only allowing a maximum of 10 units of flow to be sent along that particular path. If we finally look at $(4, t)$, we can see that it has several paths from $s - t$ that it is a part of, but while this arc can receive a total of 18 units of flow from $(2, 4)$ and $(3, 4)$, it only has a capacity of 8. Therefore, we can also view this as a bottleneck arc. Now if we pretend to send flow through the system in an attempt to maximize the capacity along each of these arcs, we will get something like this: $f(S, T) = \{10, 13, 5, 3, 10, 8, 5, 10\}$, where the flow here is in alphabetical order.

You may notice that we still have residual capacity on arcs: $(s, 3)$ with 1 unit residual capacity, $(3, 4)$ with 10 units of residual capacity, and $(5, t)$ with 2 units of residual capacity. Starting with the lowest residual capacity, if we increase flow on $(s, 3)$ by 1, we will see a will see an increase in flow on $(3, 4)$ to 4 units, but we will be violating the flow conservation constraint on 4! That is because of the following: $(x_{4,t}) - (x_(3, 4) + x_(2, 4)) = 1 > 0$! That means that we cannot increase flow anywhere in the graph and, thus, we are optimal. While this is by no means a theorem thus far, it will soon transform into the *Max-Flow/Min-Cut Theorem*, where the minimum cut size in a graph is equal to the maximum flow. This will be explored later in this section.

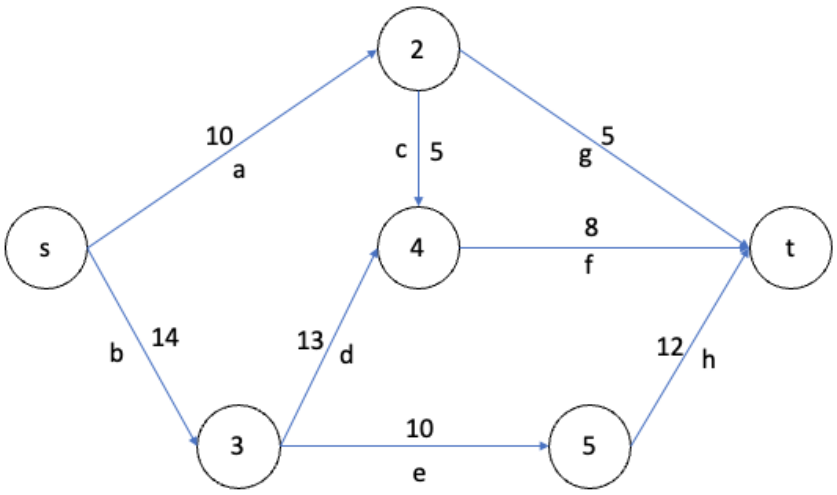

Figure 4: Maximum Flow - Example Problem

Now that we've explored how a maximum flow problem works and under-

stand what a minimum cut is, lets explore an algorithm to solve these types of problems without brute force.

3.1 The Ford-Fulkerson (Augmenting Paths) Algorithm

The Ford-Fulkerson Algorithm is a very effective way of solving the maximum flow problem. The algorithm relies heavily on the construction of a **residual graph** and the idea of an **augmenting path**. We will begin by defining the residual graph.

The residual graph is effectively the storage of all residual capacity left in the graph with the goal of effectively *undoing* un-optimal choices made by our algorithm. How this works will become more clear as the algorithm is explained. We will from hereon refer to the residual graph as G' and the original graph as G. We will construct our residual graph as follows:

1. All nodes in the original graph carry over to the residual graph unchanged.
2. Pick an arc (i, j) in G. Construct an arc (j, i) in G' with capacity $u'_{ji} = x_{ij}$
3. For the same arc (i, j) in G, construct an arc (i, j) in G' with capacity $u'_{ij} = u_{ij} - x_{ij}$
4. Flow is irrelevant in the residual graph, so we will not have an x_{ij} variable on these newly constructed arcs.

The second step may seem complicated, but it is exactly what allows for the *undoing* that I mentioned before. By constructing an arc in the residual graph (j, i) directionally opposing an arc with flow in the original graph (i, j), and defining the capacity along this new arc as $u'_{ji} = x_{ij}$, we are effectively saying "we can un-send the flow that already exists on (i, j). Pretty cool! For the third step listed above, it is clear that this newly constructed arc (i, j) in G' effectively represents the remaining capacity along (i, j) in G.

For example, lets say we have an arc (i, j) in G has capacity $u_{ij} = 3$ and current flow $x_{ij} = 2$. This arc has a residual capacity of 1 and a flow of 2. Therefore, in the residual graph G', we will have constructed an arc (i, j) with capacity $u'_{ij} = 1$, and an additional arc (j, i) with capacity $u'_{ij} = 2$. Pretty intuitive right!

From hereon, the only thing that will change in the original graph with be the flow x_{ij} along each arc, and the only thing that will change in the residual graph will be the capacity u'_{ij} along each arc.

Now how does this help us? This is where augmenting paths come into play.

Definition 2 (Augmenting Path)**.** *A path of arcs from the s to t with unused non-zero flow, or positive residual capacity.*

That's pretty straight forward. It will hold true that if there exists an augmenting path, then there must be some additional flow that can but sent through the graph and, thus, we are not yet optimal. It might be relatively easy

to pick up where this is going at this point. The Ford-Fulkerson Algorithm relies on a combination of both augmenting paths and the construction of a residual graph to find the maximum flow in a given graph G!

The algorithm operates as follows:

1. G begins with 0 flow. Construct G' (only thing that should be constructed here in G' are the arcs (i, j) with $u'_{ij} = u_{ij}$ because of 0 flow).
2. Find an augmenting path in the G'. If non exist, maximum flow has been reached. *
3. Define $\overline{U}_c$ as the minimum residual capacity in the augmenting path.
4. Increase flow along the path in G by $\overline{U}_c$. If an arc in the augmenting path (in G') is directionally opposing an arc in G, decrease the flow along the arc in G by $\overline{U}_c$.
5. Return to step 2.

* It is extremely important to note the time complexity of this particular algorithm is largely dependent upon how you go about searching for

Now that we understand how to run the Ford-Fulkerson Algorithm, lets try to do it on figure 4.

We begin with the following original and residual graphs:

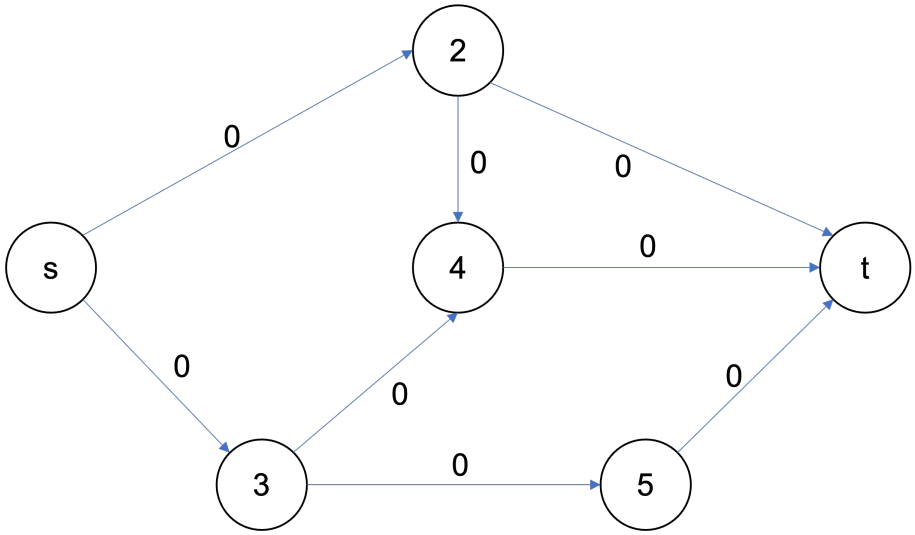

Figure 5: Ford-Fulkerson Example Step 1 - Original Graph, G

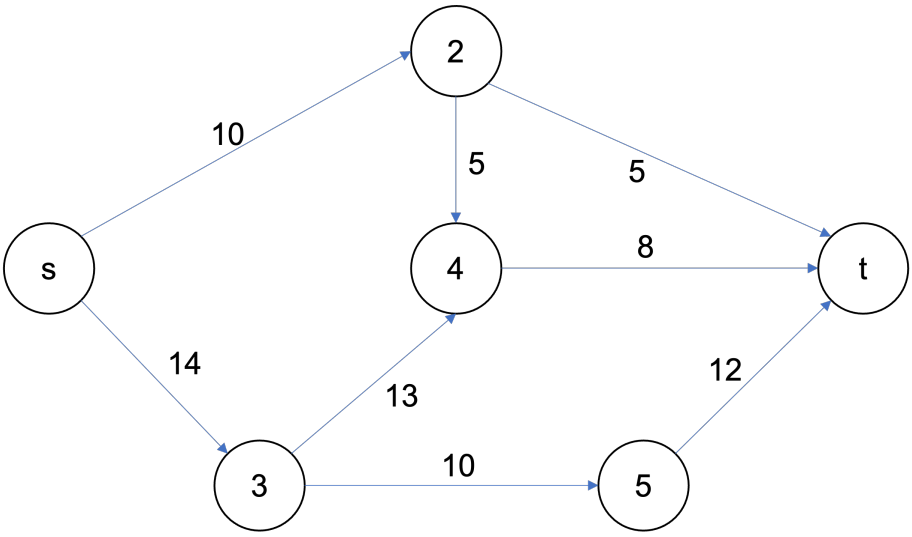

Figure 6: Ford-Fulkerson Example Step 1 - Residual Graph, G'

We now find an augmenting path through the G' and send flow through it in the G. Lets say we find the path $\{(s, 2), (2, t)\}$. The minimum residual capacity along that path is 5, so we will send 5 units of flow along it in G. We also update our residual graph accordingly.

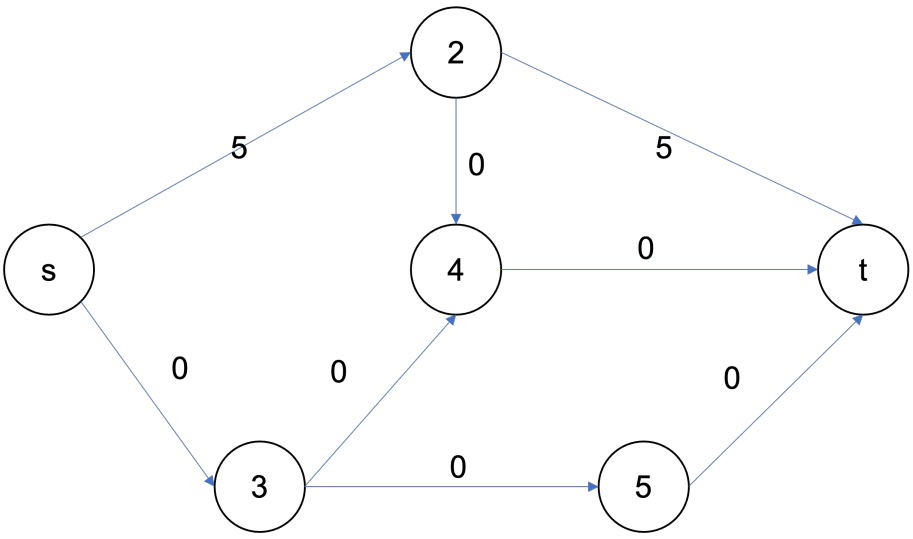

Figure 7: Ford-Fulkerson Example Step 2 - Original Graph, G

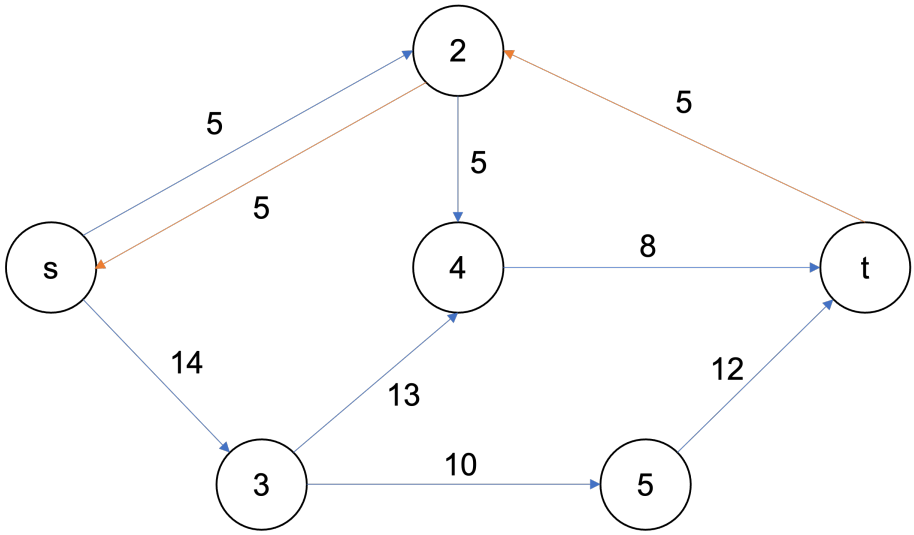

Figure 8: Ford-Fulkerson Example Step 2 - Residual Graph, G'

Now, we again search for an augmenting path through the G' and send flow through it in the G. Lets say we find the path $\{(s,3),(3,4),(4,t)\}$. The minimum residual capacity along that path is 8, so we will send 8 units of flow along it in G. We also update our residual graph accordingly.

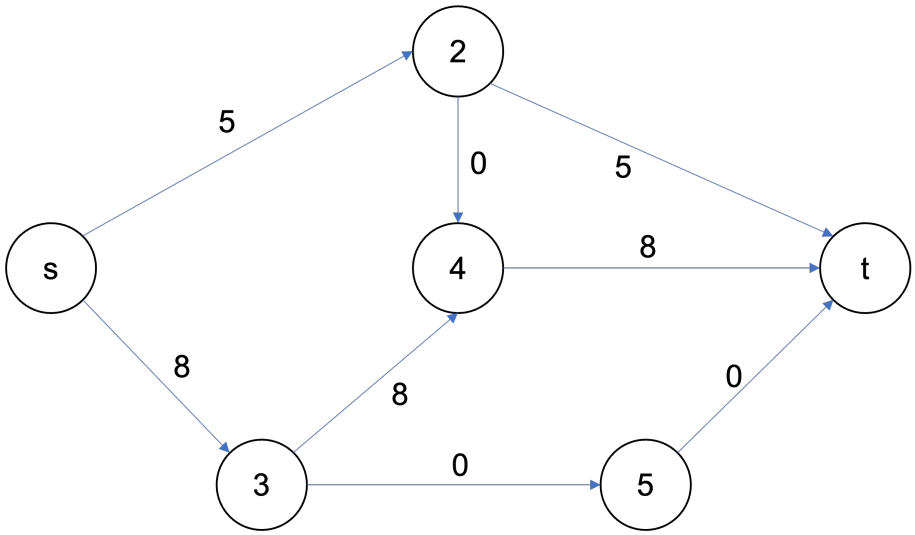

Figure 9: Ford-Fulkerson Example Step 3 - Original Graph, G

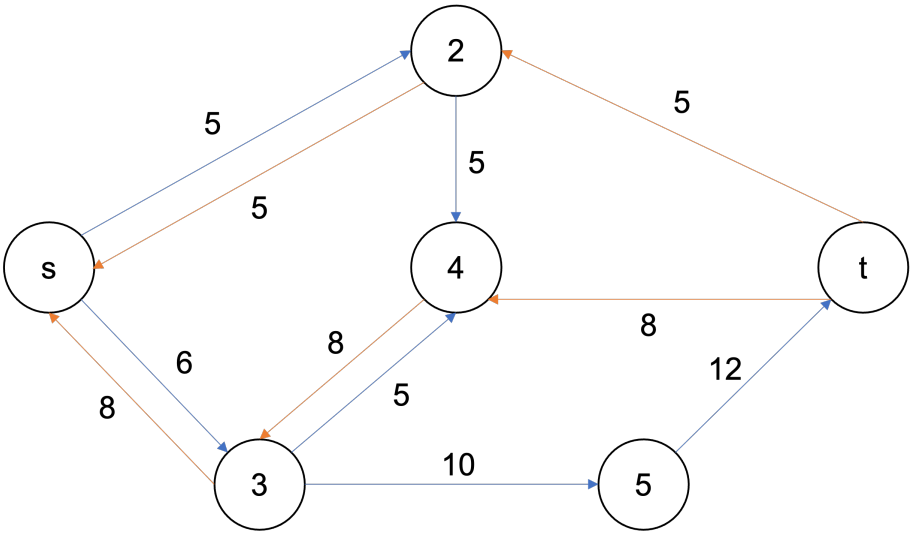

Figure 10: Ford-Fulkerson Example Step 3 - Residual Graph, G'

Again, we again search for an augmenting path through the G' and send flow through it in the G. Lets say we find the path $\{(s,3),(3,5),(5,t)\}$. The minimum residual capacity along that path is 6, so we will send 6 units of flow along it in G. We also update our residual graph accordingly.

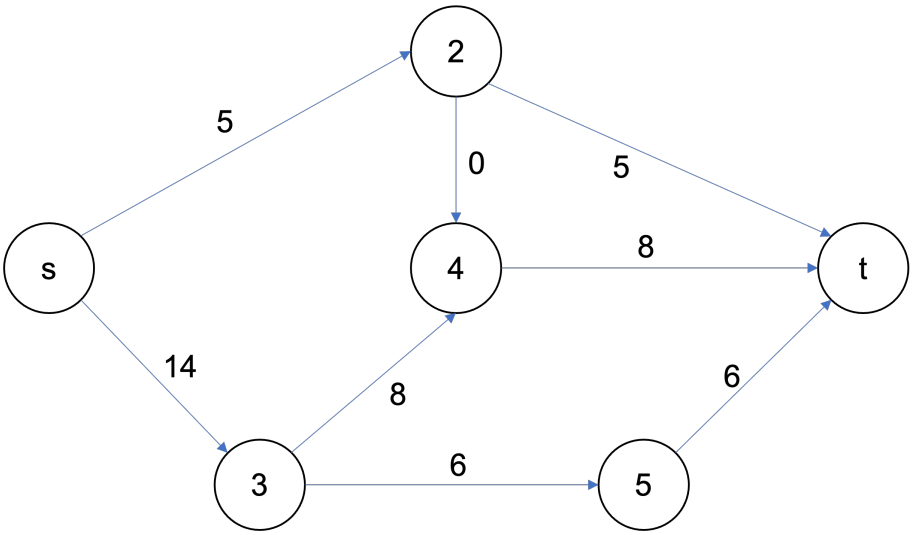

Figure 11: Ford-Fulkerson Example Step 4 - Original Graph, G

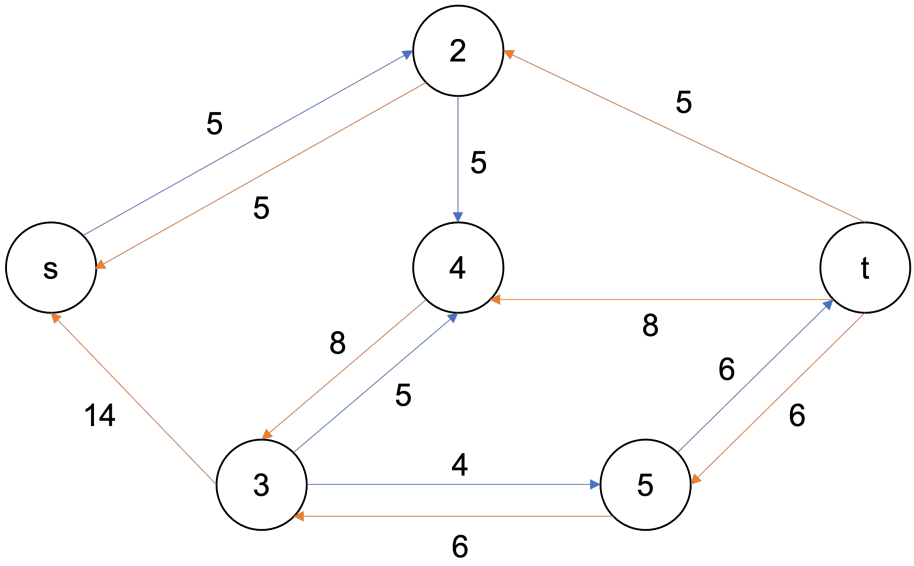

Figure 12: Ford-Fulkerson Example Step 4 - Residual Graph, G'

We once more search for an Augmenting Path through G'. Surprisingly, here we can only find the path $\{(s,2),(2,4),(4,3),(3,5),(5,t)\}$, which contains the reverse arc $(4,3)$. The minimum residual capacity through this path is 4, so we send 4 units of flow through the path on G and update the residual graph accordingly.

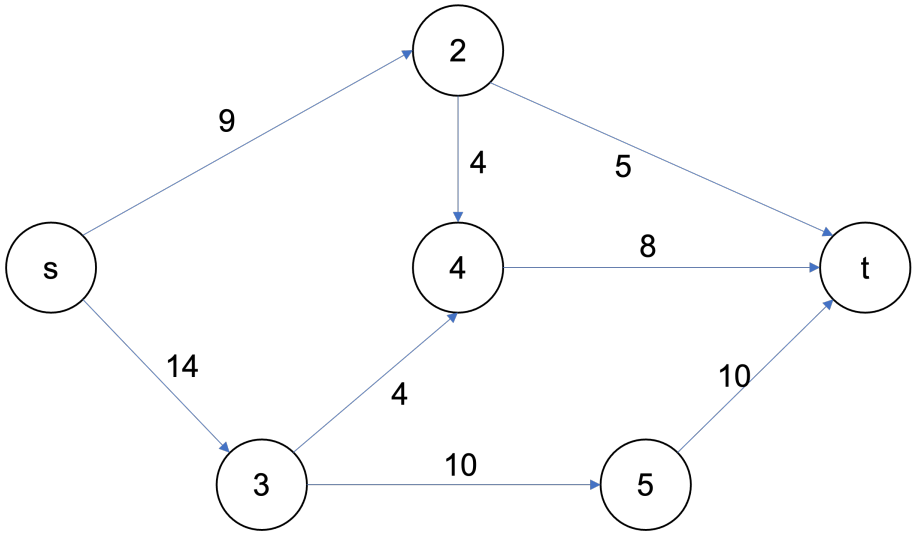

Figure 13: Ford-Fulkerson Example Step 5 - Original Graph, G

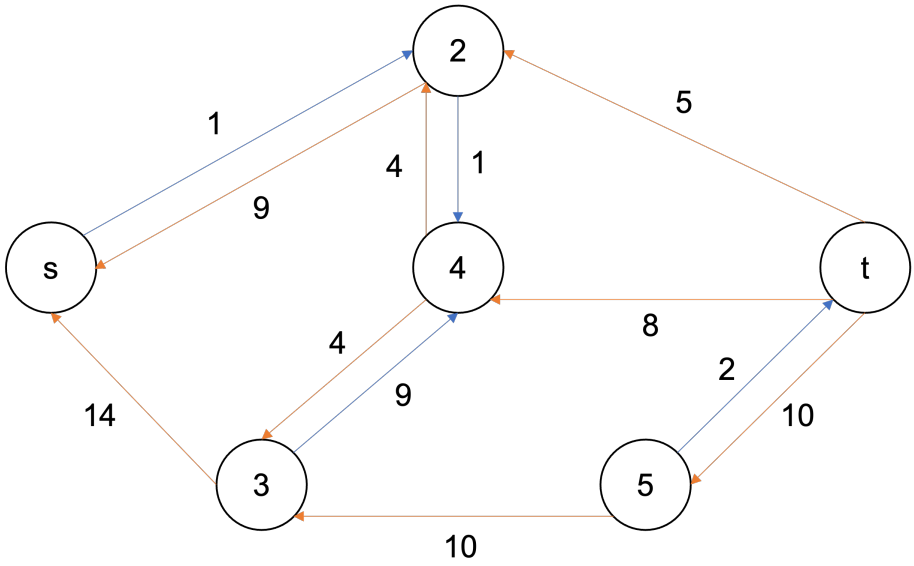

Figure 14: Ford-Fulkerson Example Step 5 - Residual Graph, G'

When we search for an augmenting path through G' now, none can be found. Therefore, we have achieved the maximum flow! Our maximum flow can be found by summing the inflows to the terminal node in G, which gives us a maximum flow of 23!

Theorem 3.1 (Max-Flow/Min-Cut)**.** *an example*

4 The Minimum Cost/Maximum Flow Problem

The Minimum Cost/Maximum Flow Problem combines components from both of the problems we have addressed in this paper thus far. Taking the distance/-cost component from the shortest path problem and the capacity component from the maximum flow problem, this new problem can tend to be quite intimidating. However, by utilizing the various algorithms that we have already explored, we can create an algorithm that can easily tackle such problems.

The minimum cost/maximum flow problem can be reduced to having the same two components that we've dealt with throughout this paper: a set of directional arcs or edges, E; and a set of nodes through which these arcs pass, V. For each arc, we will have two parameters: a capacity denoted u_{ij}; and a cost denoted c_{ij}. In this case, our cost will be considered a **cost per unit flow**, with the net cost on an arc being $x_{ij} * c_{ij}$. Each arc will additionally have a flow variable denoted x_{ij}.

Finally, where as in the previous problems we had one source node and one terminal node, here we will have nodes that supply the system with flow. Each node with have a supply b_i.

1. if $b_i > 0$, then the node will be defined as a source (supplies flow to the network)
2. if $b_i < 0$, then the node will be defined as a sink (demands flow from the network)
3. if $b_i = 0$, then the node will be defined as a transhipment/flow conserving node

In this problem, we must generate a feasible flow, which is a flow in which capacity is not broken along any arc, and all supply and demand is met on the source and sink nodes. The constraints from the previous problems will be modified slightly to meet these goals. The capacity constraint (3) will remain the same. However, the flow conservation constraint will change slightly to account for the addition of source and sink nodes (4). The modified constraints from the previous problems are below:

$$0 \leq x_{ij} \leq u_{ij} \; \forall \; i, j \epsilon V \tag{3}$$

$$\sum_{(i,j)\epsilon E} x_{ij} - \sum_{(j,i)\epsilon E} x_{ji} = b_i \; \forall \; i, j \epsilon V \tag{4}$$

We will also have an additional constraint on supply, that being that the sum of all supply in the system must be zero. If this constraint did not exist, then no flow can exist within the system. This is because when we add up the all flow conservation constraints, you will always end up with 0 on the left hand side. Therefore, you must end up with 0 on the right hand side. It is important to note that such a constraint is necessary, but not sufficient to achieve a feasible flow. The constraint described is below:

$$\sum_{i} b_i = 0 \; \forall \; i \epsilon V \tag{5}$$

There are no other limitations with this algorithm (besides integer programming and simple paths), and therefore negative cost arcs are allowed.

The goal of this problem is to find the maximum flow through a given graph while also minimizing cost, and thus our objective function is $\sum_{(i,j)\epsilon E} c_{ij} * x_{ij}$.

Now that we understand the fundamentals of the problem, lets go about constructing an algorithm to solve such a problem.

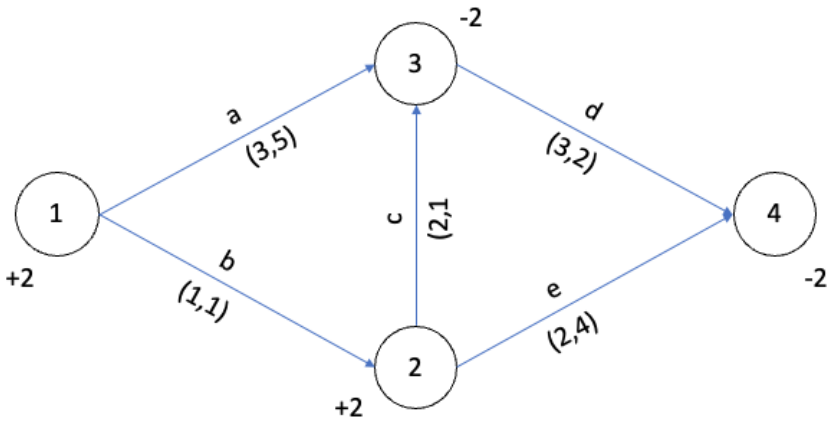

Figure 15: Minimum Cost/Maximum Flow - Example Problem

4.1 Negative Cycle Cancelling Algorithm

The Negative Cycle Cancelling algorithm is a comprehensive algorithm for solving minimum cost/maximum flow problems. The algorithm can be split into two fundamental steps, which we will be going into independently:

1. Feasible Flow Generation
2. Negative Cycle Detection

The algorithm with the above parts not explicated on is as follows:

1. Generate a feasible flow $\mathbf{X}$ (see 4.1.1)
2. Generate residual graph G'
3. Detect a negative cycle C in G' where each arc in C has positive residual capacity and the cycle has net negative cost, i.e. $\sum_{i,j \epsilon C} c_{ij}$. If no such cycle exists, optimal solution has been found (see 4.1.2)
4. Define $\overline{U}_c$ as the minimum residual capacity within this cycle
5. Update $\mathbf{X}$ by adding $\overline{U}_c$ to the flow along each arc in C that exists in the original graph G and subtracting $\overline{U}_c$ from the flow along each arc in G whose mirror image exists in C.

While this algorithm might sound intensive, we have already done a lot of the legwork to get it working! In the following sections, each component of this algorithm will be explained in detail and a final algorithm will come to form.

4.1.1 Feasible Flow Generation

The only limitation on a feasible flow is that it must satisfy all of the constraints for the problem. Because of this, any algorithm that generates a valid flow of any magnitude is utilizable. This flow need not be in consideration of costs.

In our case, it will likely be most helpful to generate a maximum flow in the graph and work backward from there. To do this, we will be utilizing a slightly modified version of the Ford-Fulkerson algorithm that we have already explored! The algorithm can be reduced to the following:

1. Add two nodes to the graph, calling them s and t
2. Add arcs from s to all nodes in the graph with $b_i > 0$

   Define $u_{si} = b_i \; \forall \; is.t.b_i > 0$
3. Add arcs from t to all nodes in the graph with $b_i < 0$

   Define $u_{si} = -b_i \; \forall \; is.t.b_i < 0$
4. RUN FORD-FULKERSON
5. Update flow on G remove all added nodes and arcs from G

The reason why we construct these two new nodes in the graph is because Ford-Fulkerson does not know how to handle supply and sink nodes, nor does it know how to handle multiple source nodes and multiple destination nodes. Following from the example figure **??**, after adding these nodes and arcs, we are left with the following graph:

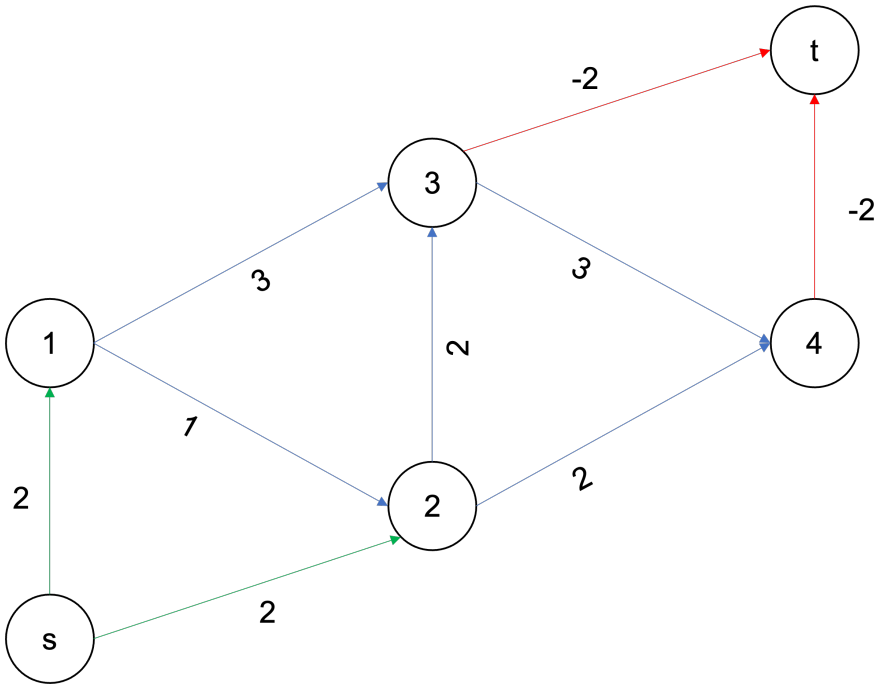

Figure 16: Negative Cycle Cancelling - Feasible Flow Generation 1
Numbers on arcs correspond to their capacity

We will node be explaining how to run Ford-Fulkerson as we have already discussed this in detail in section 3.1. However, the returned graph will be the following:

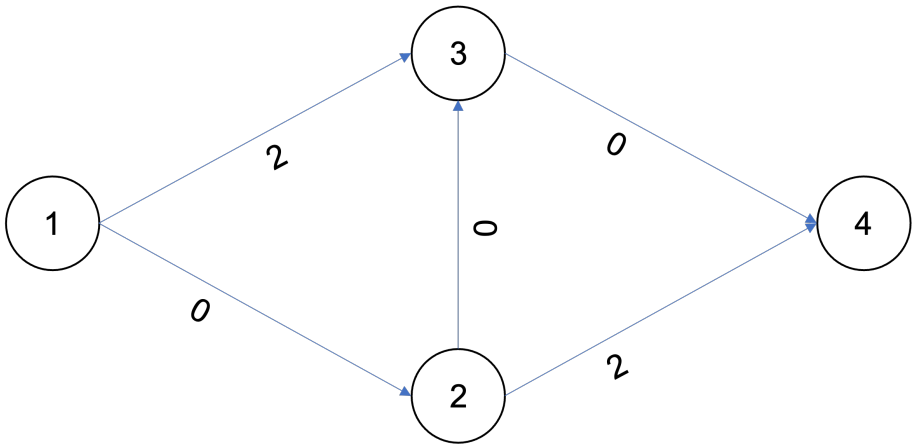

Figure 17: Negative Cycle Cancelling - Feasible Flow Generation 2
Numbers on arcs correspond to their flow

4.1.2 Negative Cycle Detection

Negative Cycle Detection can prove to be quite difficult, namely because most algorithms lack the capability of catching them. For this exact reason, Dijkstra's Algorithm is incapable of detecting a negative cycle because it closes the loop on all previously marked nodes, rendering an incorrect shortest path (see 2.1 for a more detailed explanation). Because of this, Bellman-Ford appears to be the most widely used and most effective process (even though it is higher time complexity than Dijkstra's Algorithm).

However, Bellman-Ford has to be slightly modified in order to achieve such a feat. The following is the modified Bellman-Ford Algorithm:

1. Generate a residual graph (does not contain new nodes from the previous section)
2. Establish a pointer parameter p_i for each node with initial condition $p_i =$ **null**, and a predicted distance variable d_i for each node with initial condition $d_i = +\infty$
3. set $d_i = 0 \; \forall \; is.t.b_i > 0$
4. Relax all d_i $|V| - 1$ times through the following:

   For each arc (i, j):

   if $d_i + c_{ij} < d_j$, then $d_j = d_i + c_{ij}$ and $p_j = i$
5. Repeat search for one more iteration: For each arc (i, j)

   if $d_i + c_{ij} < d_j$ AND $u_{ij} > 0$, then negative cycle detected. Store j as in the negative cycle. Do not update variables of j. If no such cycle detected, then TERMINATE - optimal flow achieved
6. Pull parent nodes from the stored j above to find all arcs in the negative cycle
7. Test each arc in the cycle for positive residual capacity. If not all arcs have positive residual capacity, then do nothing and return to step 1
8. Define $\overline{U}_c$ as the minimum residual capacity within this cycle
9. Update the feasible flow from Ford-Fulkerson $\mathbf{X}$ by adding $\overline{U}_c$ to the flow along each arc in C that exists in the original graph G and subtracting $\overline{U}_c$ from the flow along each arc in G whose mirror image exists in C.
10. return to step 1

And that leaves us with our Minimum Cost/Maximum Flow solution! Complicated, but all of these components have already been explored in great detail with the shortest path and maximum flow problems. All we had to do was put it all together!

It seems unlikely that a example of this algorithm would be useful in practice for understanding how the algorithm works. Rather, a detailed explanation of a code example of the algorithm can be seen in the next section.

5 Programs for Solving Network Flow Problems

Every algorithm discussed in this paper can be programmed to complete their desired tasks. In this section, I have programmed each respective algorithm in Java and have provided a detailed explanation for how they work. The full code for each of the below algorithms can be found in the Appendex section, or can be downloaded from GitHub (link provided below and also can be found on title page).

5.1 The Shortest Path Problem

5.1.1 Dijkstra's Algorithm

To begin, we will construct three classes: Node, Arc, and Graph. Our Node and Arc classes are relatively simple, containing a singular constructor and a set of instance variables that are publicly accessible for easy access(for the case of this problem, we will not be concerning ourselves with privacy or security, so public instance variables will be OK).

We will define a Node by an integer name, an integer predicted distance from the source node, a pointer node to its parent in the shortest distance path, and a Boolean variable for marking that is initialized to false on construction. We will define an arc by a name, a pair of nodes (n1 and n2), and a distance/cost parameter, all of which will be initialized by our constructor.

```
public class Node {
   public int name;
   public Node parent;
   public int dist;
   public boolean marked;
    public Node(int name) {
        this.name = name;
        this.marked = false;
        this.parent = null;
        this.dist = Integer.MAX_VALUE;
    }
}

public class Arc {
    public char name;
    public Node n1;
    public Node n2;
    public int cost;

    public Arc(char name, Node n1, Node n2, int c) {
        this.name = name;
        this.n1 = n1;
        this.n2 = n2;
        this.cost = c;
    }
```

```
}
```

Listing 1: Dijkstra - Node and Arc Classes

Our graph class will hold all of the components of the original graph in one place as well as all of the functions necessary for the algorithm. Instance variables include the following:

1. A Node to hold the source node
2. A Node to hold the terminal node
3. A Set of all nodes in the graph, including source and terminal
4. A Set of all arcs in the graph
5. A static String holding the path to the text representation in the graph in the current directory

```
public class Graph {
    public Node start = null;
    public Node terminal = null;
    public Set<Node> nodes = new HashSet<>();
    public Set<Arc> arcs = new HashSet<>();

    public static String filename = "[graph txt file name]";
    public static String SRC = "src/" + filename + ".txt";
```

Listing 2: Dijkstra - Graph Class Instance Variables

The graph class does not hold a constructor, rather the graph is initialized by an initialization function that pulls all of the graph's data from the text source file into the instance variables. This function is largely dependent on how your text file is written, so it will not be included in this explanation.

Dijkstra's Algorithm returns the integer value distance of the shortest path it finds within the graph rather than the path itself. Because of this, we will construct two separate functions, one to find the shortest path and return its distance (Dijkstra's Algorithm), and a helper function to return the node path of this shortest path from the source node to the terminal node (this function will not be included in this explanation).

Dijkstra's algorithm begins by un-marking all nodes in the graph and defining their initial predicted shortest distance in the graph as infinity. It then iterates through all arcs in the graph under the condition that there still exist unmarked nodes and beginning with the node that minimizes distance (hereon referred to as the selected node), and relaxing the end node of each arc whose beginning node is equal to the selected node by updating shortest distance and setting its parent pointer to the selected node (i.e. $if\ d_j > d_i + c_ij,\ then\ d_j = d_i + c_ij\ AND\ p_j = i$). Once each adjacent node to the selected node is updated, the selected node is marked and the algorithm continues until all nodes

are marked. Once all nodes are marked, an optimal shortest path has been found, and the algorithm pulls and returns the predicted distance from the terminal node.

Each step listed above is included in comments in the code below.

```
public int dijkstra() {
    // 1) create unmarked set and fill with all nodes (all initially
        unmarked)
    Set<Node> unmarked = new HashSet<>();
    for (Node node : nodes) {
        unmarked.add(node);
        node.dist = Integer.MAX_VALUE;
    }

    // 2) set distance for start node to 0, distance to itself is 0
    start.dist = 0;

    // 3) while there are unmarked nodes, do the following...
    while (!unmarked.isEmpty()) {

        // 4) find unmarked node with smallest predicted distance
            (initially will be our
        // starting node)
        Node cur = null;
        int min = Integer.MAX_VALUE;
        for (Node node : unmarked) {
            if (node.dist < min) {
                min = node.dist;
                cur = node;
            }
        }

        // 5) iterate through all arcs to find nodes neighboring our
            selected node and
        // test shortest distance
        for (Arc arc : arcs) {
            if (arc.n1.equals(cur) && unmarked.contains(arc.n2)) {
                if ((arc.n2.dist > cur.dist + arc.cost) && cur.dist !=
                    Integer.MAX_VALUE) {
                    arc.n2.dist = cur.dist + arc.cost;
                    arc.n2.parent = cur;
                }
            }
        }

        // 6) mark cur
        unmarked.remove(cur);
        cur.marked = true;
    }
```

```
        // 7) pull shortest distance from the terminal node
        for (Node node : nodes) {
            if (node.equals(terminal)) {
                return node.dist;
            }
        }
        return 0;
    }
```

Listing 3: Dijkstra - Dijkstra's Algorithm

Our main method will simply run each of the above functions and print their output.

```
    public static void main(String[] args) throws Exception {
        Graph g = new Graph();
        g.initGraph();
        System.out.println("Shortest distance with Dijkstra: " +
            g.dijkstra());
        System.out.println("Shortest Path with Dijkstra: " +
            Arrays.toString(g.getShortestPath()));
    }
```

Listing 4: Dijkstra - Main Method

The output of the above code run on the example graph in figure 2 (see 2.1) is the following:

```
Shortest distance with Dijkstra: 2
Shortest Path with Dijkstra: [1, 2, 6]
```

The full code for Dijkstra's Algorithm can be found in Appendix A.

5.1.2 The Bellman-Ford Algorithm

We will begin with the same class structure as in Dijkstra's Algorithm (see 2.3.1), with the only change being the introduction of the Bellman-Ford Algorithm function. The Bellman-Ford Algorithm is much simpler than Dijkstra's algorithm in structure with the trade off of having a significantly higher time complexity due to its recursive nature (see [INSERT] for explanation). Like Dijkstra, the algorithm only returns the integer value shortest distance and not the shortest path itself, so an additional helper function is required to backtrack from the terminal node to the source node and return the shortest path.

The algorithm begins by setting the distance prediction of the source node to zero. The algorithm proceeds by relaxing each node in the network using the same criteria used in Dijkstra ($if\ d_j > d_i + c_ij,\ then\ d_j = d_i + c_ij$), completing only $|nodes| - 1$, or one less than the cardinality of the node set, iterations. Because this should achieve the optimal shortest distance in the graph, any

other shorter distance found indicates a negative cycle. Thus, the algorithm completes one more iteration to determine optimality. The algorithm finally pulls and returns the distance value from the terminal node.

Each step listed above is included in comments in the code below.

```
public int bellmanFord() {
    // 1) set distance for start node to 0, distance to itself is 0
    start.dist = 0;


    // 2) Relax each index in our node set by updating shortest
        distance to each
    // node |nodes| - 1 times
    for (int i = 1; i < nodes.size(); i++) {
        for (Arc arc : arcs) {
            if ((arc.n2.dist > arc.n1.dist + arc.cost) &&
                (arc.n1.dist != Integer.MAX_VALUE)) {
                arc.n2.dist = arc.n1.dist + arc.cost;
                arc.n2.parent = arc.n1;
            }
        }
    }

    // 3) Repeat to detect any negative cycles, return 0 if detected
        (step 2 should
    // be optimal, so if there is another shorter path, there must
        be a negative
    // cycle)
    for (Arc arc : arcs) {
        if ((arc.n2.dist > arc.n1.dist + arc.cost) && (arc.n1.dist !=
            Integer.MAX_VALUE)) {
            return 0;
        }
    }
    // 4) pull shortest distance from the terminal node
    for (Node node : nodes) {
        if (node.name == terminal.name) {
            return node.dist;
        }
    }
    return 0;
}
```

Listing 5: Bellman-Ford - Bellman-Ford Algorithm

The output of the above code run on the example graph in figure 2 (see 2.1) is the following:

```
Shortest distance with Bellman-Ford: 2
Path with Bellman-Ford: [1, 2, 6]
```

The full code for the Bellman-Ford Algorithm can be found in Appendix B.

6 Conclusion

7 References

Appendices

A Dijkstra's Algorithm

```
/* Dijkstra's Algorithm for Shortest Path
 */

import java.util.*;
import java.io.*;


public class Arc {
    public char name;
    public Node n1;
    public Node n2;
    public int cost;

    public Arc(char name, Node n1, Node n2, int c) {
        this.name = name;
        this.n1 = n1;
        this.n2 = n2;
        this.cost = c;
    }
}

public class Node {
   public int name;
   public Node parent;
   public int dist;
   public boolean marked;

    public Node(int name) {
        this.name = name;
        this.marked = false;
        this.parent = null;
      this.dist = Integer.MAX_VALUE;
    }
}
```

```
public class Graph {
    public Node start = null;
    public Node terminal = null;
    public Set<Node> nodes = new HashSet<>();
    public Set<Arc> arcs = new HashSet<>();

    // CHANGE "filename" TO SELECT THE TEST GRAPH
    public static String filename = "testGraph1"; // "testGraph2";
    public static String SRC = "dijkstra-and-bellman-ford/src/" +
        filename + ".txt";

    /******** INITIALIZING GRAPH ********/
    /*
     * Returns: null
     */
    public void initGraph() throws FileNotFoundException {
        File file = new File(SRC);
        Scanner sc = new Scanner(file);
        int iter = 0;
        Set<Integer> nodesNames = new HashSet<Integer>();
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            String[] split = line.split(" ");

            if (iter == 0) {
                int first = Integer.parseInt(split[0]);
                nodesNames.add(first);
                Node start = new Node(first);
                nodes.add(start);
                this.start = start;
                int second = Integer.parseInt(split[1]);
                nodesNames.add(second);
                Node terminal = new Node(second);
                nodes.add(terminal);
                this.terminal = terminal;
                iter++;
            } else {
                char arcName = split[0].charAt(0);
                int first = Integer.parseInt(split[1]);
                int second = Integer.parseInt(split[2]);
                int cost = Integer.parseInt(split[3]);
                Node n1 = null;
                Node n2 = null;
                if (!nodesNames.contains(first)) {
                    n1 = new Node(first);
                    nodesNames.add(first);
                    nodes.add(n1);
                } else {
                    for (Node cur : nodes) {
                        if (cur.name == first) {
```

```
                    n1 = cur;
                }
            }
        }
        if (!nodesNames.contains(second)) {
            n2 = new Node(second);
            nodesNames.add(second);
            nodes.add(n2);
        } else {
            for (Node cur : nodes) {
                if (cur.name == second) {
                    n2 = cur;
                }
            }
        }
        Arc arc = new Arc(arcName, n1, n2, cost);
        arcs.add(arc);
    }
  }
}

/******** DIJKSTRA'S ALGORITHM ********/
/*
 * Returns: integer, the shortest distance from source node to
     terminal node
 */
public int dijkstra() {
    // 1) create unmarked set and fill with all nodes (all initially
        unmarked)
    Set<Node> unmarked = new HashSet<>();
    for (Node node : nodes) {
        unmarked.add(node);
        node.dist = Integer.MAX_VALUE;
    }

    // 2) set distance for start node to 0, distance to itself is 0
    start.dist = 0;

    // 3) while there are unmarked nodes, do the following...
    while (!unmarked.isEmpty()) {

        // 4) find unmarked node with smallest predicted distance
            (initially will be our
        // starting node)
        Node cur = null;
        int min = Integer.MAX_VALUE;
        for (Node node : unmarked) {
            if (node.dist < min) {
                min = node.dist;
                cur = node;
```

```
            }
        }

        // 5) iterate through all arcs to find nodes neighboring our
            selected node and
        // test shortest distance
        for (Arc arc : arcs) {
            if (arc.n1.equals(cur) && unmarked.contains(arc.n2)) {
                if ((arc.n2.dist > cur.dist + arc.cost) && cur.dist !=
                    Integer.MAX_VALUE) {
                    arc.n2.dist = cur.dist + arc.cost;
                    arc.n2.parent = cur;
                }
            }
        }

        // 6) mark cur
        unmarked.remove(cur);
        cur.marked = true;
    }

    // 7) pull shortest distance from the terminal node
    for (Node node : nodes) {
        if (node.equals(terminal)) {
            return node.dist;
        }
    }
    return 0;
}

/******** GETTING SHORTEST PATH ********/
/*
 * Returns: integer array with shortest path, beginning with source
    node and
 * ending with terminal node
 */
public int[] getShortestPath() {
    List<Node> backpath = new ArrayList<Node>();
    Node term = null;
    for (Node node : nodes) {
        if (node.equals(this.terminal)) {
            term = node;
        }
    }
    backpath.add(term);
    while (!term.equals(start)) {
        term = term.parent;
        backpath.add(term);
    }
```

```
        int[] path = new int[backpath.size()];
        Iterator<Node> iter = backpath.iterator();
        int i = 0;
        while (iter.hasNext()) {
            path[backpath.size() - 1 - i] = iter.next().name;
            i++;
        }
        return path;
    }

    public static void main(String[] args) throws Exception {
        Graph g = new Graph();
        g.initGraph();
        System.out.println("Shortest distance with Dijkstra: " +
            g.dijkstra());
        System.out.println("Shortest Path with Dijkstra: " +
            Arrays.toString(g.getShortestPath()));
    }
}
```

B Bellman-Ford Algorithm

```
/* Bellman-Ford Algorithm for Shortest Path
 */

import java.util.*;
import java.io.*;


public class Arc {
    public char name;
    public Node n1;
    public Node n2;
    public int cost;

    public Arc(char name, Node n1, Node n2, int c) {
        this.name = name;
        this.n1 = n1;
        this.n2 = n2;
        this.cost = c;
    }
}

public class Node {
   public int name;
   public Node parent;
   public int dist;
```

```
   public boolean marked;

    public Node(int name) {
        this.name = name;
        this.marked = false;
        this.parent = null;
      this.dist = Integer.MAX_VALUE;
    }
}

public class Graph {
    public Node start = null;
    public Node terminal = null;
    public Set<Node> nodes = new HashSet<>();
    public Set<Arc> arcs = new HashSet<>();

    // CHANGE "filename" TO SELECT THE TEST GRAPH
    public static String filename = "testGraph1"; // "testGraph2";
    public static String SRC = "dijkstra-and-bellman-ford/src/" +
        filename + ".txt";

    /******** INITIALIZING GRAPH ********/
    /*
     * Returns: null
     */
    public void initGraph() throws FileNotFoundException {
        File file = new File(SRC);
        Scanner sc = new Scanner(file);
        int iter = 0;
        Set<Integer> nodesNames = new HashSet<Integer>();
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            String[] split = line.split(" ");

            if (iter == 0) {
                int first = Integer.parseInt(split[0]);
                nodesNames.add(first);
                Node start = new Node(first);
                nodes.add(start);
                this.start = start;
                int second = Integer.parseInt(split[1]);
                nodesNames.add(second);
                Node terminal = new Node(second);
                nodes.add(terminal);
                this.terminal = terminal;
                iter++;
            } else {
                char arcName = split[0].charAt(0);
                int first = Integer.parseInt(split[1]);
                int second = Integer.parseInt(split[2]);
```

```
            int cost = Integer.parseInt(split[3]);
            Node n1 = null;
            Node n2 = null;
            if (!nodesNames.contains(first)) {
                n1 = new Node(first);
                nodesNames.add(first);
                nodes.add(n1);
            } else {
                for (Node cur : nodes) {
                    if (cur.name == first) {
                        n1 = cur;
                    }
                }
            }
            if (!nodesNames.contains(second)) {
                n2 = new Node(second);
                nodesNames.add(second);
                nodes.add(n2);
            } else {
                for (Node cur : nodes) {
                    if (cur.name == second) {
                        n2 = cur;
                    }
                }
            }
            Arc arc = new Arc(arcName, n1, n2, cost);
            arcs.add(arc);
        }
    }
}

/******** BELLMAN-FORD ALGORITHM ********/
/*
 * Returns: integer, the shortest distance from source node to
     terminal node
 */
public int bellmanFord() {
    // 1) set distance for start node to 0, distance to itself is 0
    start.dist = 0;


    // 2) Relax each index in our node set by updating shortest
        distance to each
    // node |nodes| - 1 times
    for (int i = 1; i < nodes.size(); i++) {
        for (Arc arc : arcs) {
            if ((arc.n2.dist > arc.n1.dist + arc.cost) &&
                (arc.n1.dist != Integer.MAX_VALUE)) {
                arc.n2.dist = arc.n1.dist + arc.cost;
                arc.n2.parent = arc.n1;
```

```
            }
        }
    }

    // 3) Repeat to detect any negative cycles, return 0 if detected
        (step 2 should
    // be optimal, so if there is another shorter path, there must
        be a negative
    // cycle)
    for (Arc arc : arcs) {
        if ((arc.n2.dist > arc.n1.dist + arc.cost) && (arc.n1.dist !=
            Integer.MAX_VALUE)) {
            return 0;
        }
    }
    // 4) pull shortest distance from the terminal node
    for (Node node : nodes) {
        if (node.name == terminal.name) {
            return node.dist;
        }
    }
    return 0;
}

/******** GETTING SHORTEST PATH ********/
/*
 * Returns: integer array with shortest path, beginning with source
     node and
 * ending with terminal node
 */
public int[] getShortestPath() {
    List<Node> backpath = new ArrayList<Node>();
    Node term = null;
    for (Node node : nodes) {
        if (node.equals(this.terminal)) {
            term = node;
        }
    }
    backpath.add(term);
    while (!term.equals(start)) {
        term = term.parent;
        backpath.add(term);
    }

    int[] path = new int[backpath.size()];
    Iterator<Node> iter = backpath.iterator();
    int i = 0;
    while (iter.hasNext()) {
        path[backpath.size() - 1 - i] = iter.next().name;
        i++;
```

```
        }
        return path;
    }

    public static void main(String[] args) throws Exception {
        Graph g = new Graph();
        g.initGraph();
        System.out.println("Shortest distance with Bellman-Ford: " +
            g.bellmanFord());
        System.out.println("Shortest Path with Bellman-Ford: " +
            Arrays.toString(g.getShortestPath()));
    }
}
```

C Ford-Fulkerson Algorithm

```
import java.util.*;
import java.io.*;

public class Arc {
    public String name;
    public Node n1;
    public Node n2;
    public int u;
    public int x;

    public Arc(String name, Node n1, Node n2, int u) {
        this.name = name;
        this.n1 = n1;
        this.n2 = n2;
        this.u = u;
        this.x = 0;
    }
}

public class Node {
   public int name;

   public Node parent;

   public int minResCap;

   public Boolean viewed;

    public Node(int name) {
        this.name = name;
      this.parent = null;
```

```
      viewed = false;
      minResCap = Integer.MAX_VALUE;
    }
}


import java.util.*;

import java.io.*;

public class Graph {
    public Node start = null;
    public Node terminal = null;
    public Set<Node> nodes = new HashSet<>();
    public Set<Arc> arcs = new HashSet<>();

    // CHANGE "filename" TO SELECT THE TEST GRAPH
    public static String filename = "testGraph3"; // "testGraph2";
    public static String SRC = "src/" + filename + ".txt";

    /******** INITIALIZING ORIGINAL GRAPH ********/
    /*
     * Returns: null
     */
    public void initOriginalGraph() throws FileNotFoundException {
        File file = new File(SRC);
        Scanner sc = new Scanner(file);
        int iter = 0;
        Set<Integer> nodesNames = new HashSet<Integer>();
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            String[] split = line.split(" ");

            if (iter == 0) {
                int first = Integer.parseInt(split[0]);
                nodesNames.add(first);
                Node start = new Node(first);
                nodes.add(start);
                this.start = start;
                int second = Integer.parseInt(split[1]);
                nodesNames.add(second);
                Node terminal = new Node(second);
                nodes.add(terminal);
                this.terminal = terminal;
                iter++;
            } else {
                String arcName = split[0];
                int first = Integer.parseInt(split[1]);
                int second = Integer.parseInt(split[2]);
                int u = Integer.parseInt(split[3]);
```

```
            Node n1 = null;
            Node n2 = null;
            if (!nodesNames.contains(first)) {
                n1 = new Node(first);
                nodesNames.add(first);
                nodes.add(n1);
            } else {
                for (Node cur : nodes) {
                    if (cur.name == first) {
                        n1 = cur;
                    }
                }
            }
            if (!nodesNames.contains(second)) {
                n2 = new Node(second);
                nodesNames.add(second);
                nodes.add(n2);
            } else {
                for (Node cur : nodes) {
                    if (cur.name == second) {
                        n2 = cur;
                    }
                }
            }
            Arc arc = new Arc(arcName, n1, n2, u);
            arcs.add(arc);
        }
    }
}

/******** INITIALIZING RESIDUAL GRAPH ********/
/*
 * Returns: null
 */
public void initResidualGraph(Graph g) {
    this.nodes = g.nodes; // all nodes are the same
    this.start = g.start;
    this.terminal = g.terminal;
    for (Arc arc : g.arcs) {
        this.arcs.add(new Arc(arc.name, arc.n1, arc.n2, arc.u));
        // original graph arcs in residual graph have capacity equal
            to u - x on their original graph counterpart, originally
            just u because no current flow
        this.arcs.add(new Arc("r" + arc.name, arc.n2, arc.n1, 0));
        // reverse arcs in residual graph have capacity equal to x on
            the mirror arc in the original graph, originally 0
            because no original flow
    }
}
```

```
/******** helper func - findPathFromSourceToSink ********/
/*
 * Function: searches the residual graph to see if there is a
 * path from source to sink with positive risidual capacity.
 *
 * Returns: True or false if there is a path that exists
 *
 * Updates: Parent params of nodes for backtracing and MinResCap
 * for updating capacity along arcs in residual graph
 */
public boolean findPathFromSourceToSink() { // simple Bredth First
    Search Algorithm
    int minResCap = Integer.MAX_VALUE; // stores
    for (Node node : nodes) {
        node.viewed = false;
        node.parent = null;
    }

    LinkedList<Node> queue = new LinkedList<Node>();
    start.viewed = true;
    start.parent = null;
    queue.add(start);

    while (queue.size() != 0) {
        Node cur = queue.poll();
        for (Arc arc : arcs) {
            if (arc.n1.equals(cur) && !arc.n2.viewed && arc.u > 0) {
                if (arc.u < minResCap) {
                    minResCap = arc.u; //setting minimum residual
                        capacity to be stored in terminal node
                }
                arc.n2.parent = arc.n1;

                if (arc.n2.equals(terminal)) {
                    arc.n2.minResCap = minResCap;
                    return true; //if terminal node has been found,
                        there exists an augmenting path, return true
                }

                queue.add(arc.n2);
                arc.n2.viewed = true;
            }
        }
    }
    return false;
}

/******** BELLMAN-FORD ALGORITHM ********/
/*
 * Returns: integer, the maximum flow attainable in the graph
```

```
     */
    public int fordFulkerson(Graph r) {
        // 1) Find an augmenting path in the in the residual graph. If
            non exist, maximum flow has been reached.
        while (r.findPathFromSourceToSink()) {
            // 2) update residual graph by stored minimum residual
                capacity (stored in terminal node)
            for (Node cur = terminal; !cur.equals(start); cur =
                cur.parent) {
                for (Arc rarc : r.arcs) {
                    if (rarc.n1.equals(cur) && rarc.n2.equals(cur.parent))
                        {
                        rarc.u += terminal.minResCap;
                    } else if (rarc.n2.equals(cur) &&
                        rarc.n1.equals(cur.parent)) {
                        rarc.u -= terminal.minResCap;

                    }
                }
            }
        }
        //3) update all arcs in the original graph and return the
            maximum flow value
        int sum = 0;
        for (Arc arc : this.arcs) {
            for (Arc rarc : r.arcs) {
                if (arc.n1.equals(rarc.n1) && arc.n2.equals(rarc.n2)) {
                    arc.x = arc.u - rarc.u;
                    if (arc.n2.equals(terminal)) {
                        sum += arc.x;
                    }
                }
            }
        }

        return sum;
    }

    public static void main(String[] args) throws Exception {
        Graph g = new Graph();
        g.initOriginalGraph();
        Graph r = new Graph();
        r.initResidualGraph(g);
        System.out.println("Maximum Flow: " + g.fordFulkerson(r));
        for (Arc arc : g.arcs) {
            System.out.println(arc.name + ": " + arc.x);
        }
    }
}
```

D Negative Cycle Cancelling Algorithm

```
import java.util.*;
import java.io.*;

public class Node {
   public int name;
   public int supply;


   //for bellman-ford
   public int dist;
   public boolean marked;
   public Node parent; //also for ford fulkerson
   public boolean inNegCycle;

   //for ford-fulkerson
   public int minResCap; //also for bellman-ford
   public Boolean viewed;



    public Node(int name, int supply) {
        this.name = name;
      this.parent = null;
      viewed = false;
      minResCap = Integer.MAX_VALUE;
      dist = Integer.MAX_VALUE;
      this.supply = supply;
      marked = false;
      inNegCycle = false;
    }
}

public class Arc {
    public String name;
    public Node n1;
    public Node n2;
    public int u;
    public int x;
    public int c;

    public Arc(String name, Node n1, Node n2, int u, int c) {
        this.name = name;
        this.n1 = n1;
        this.n2 = n2;
        this.u = u;
        this.x = 0;
        this.c = c;
```

```
    }
}

public class Graph {
    public Set<Node> nodes = new HashSet<>();
    public Set<Arc> arcs = new HashSet<>();
    public int[] shortestPath;

    // CHANGE "filename" TO SELECT THE TEST GRAPH
    public static String filename = "testGraph1"; // "testGraph2";
    public static String SRC = "src/" + filename + ".txt";

    /******** INITIALIZING ORIGINAL GRAPH ********/
    /*
     * Returns: void
     */
    public void initGraph() throws FileNotFoundException {
        File file = new File(SRC);
        Scanner sc = new Scanner(file);
        int iter = 0;
        Set<Integer> nodesNames = new HashSet<Integer>();
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            String[] split = line.split(" ");

            if (iter == 0) { // node, supply, node, supply,...
                for (int i = 0; i < split.length; i += 2) {
                    int name = Integer.parseInt(split[i]);
                    int supply = Integer.parseInt(split[i + 1]);
                    nodesNames.add(name);
                    Node supplyNode = new Node(name, supply);
                    nodes.add(supplyNode);
                }
                iter++;
            } else if (iter == 1) { // node, supply, node, supply,...
                for (int i = 0; i < split.length; i += 2) {
                    int name = Integer.parseInt(split[i]);
                    int supply = -1 * Integer.parseInt(split[i + 1]);
                    nodesNames.add(name);
                    Node sourceNode = new Node(name, supply);
                    nodes.add(sourceNode);
                }
                iter++;
            } else { // arcname n1 n2 capacity cost
                String arcName = split[0];
                int first = Integer.parseInt(split[1]);
                int second = Integer.parseInt(split[2]);
                int u = Integer.parseInt(split[3]);
                int c = Integer.parseInt(split[4]);
                Node n1 = null;
```

```
            Node n2 = null;
            if (!nodesNames.contains(first)) {
                n1 = new Node(first, 0);
                nodesNames.add(first);
                nodes.add(n1);
            } else {
                for (Node cur : nodes) {
                    if (cur.name == first) {
                        n1 = cur;
                    }
                }
            }
            if (!nodesNames.contains(second)) {
                n2 = new Node(second, 0);
                nodesNames.add(second);
                nodes.add(n2);
            } else {
                for (Node cur : nodes) {
                    if (cur.name == second) {
                        n2 = cur;
                    }
                }
            }
            Arc arc = new Arc(arcName, n1, n2, u, c);
            arcs.add(arc);
        }
    }
}

/******** INITIALIZING RESIDUAL GRAPH ********/
/*
 * Returns: void
 */
public void initResidualGraph(Graph g) {
    // this.nodes = g.nodes;
    for (Node node : g.nodes) {
        this.nodes.add(node);
    }
    for (Arc arc : g.arcs) {
        this.arcs.add(new Arc(arc.name, arc.n1, arc.n2, arc.u -
            arc.x, arc.c));
        this.arcs.add(new Arc("r" + arc.name, arc.n2, arc.n1, arc.x,
            -1 * arc.c));
    }
}

/******** helper func - findPathFromSourceToSink ********/
/*
 * Function: searches the residual graph to see if there is a
 * path from source to sink with positive risidual capacity.
```

```
 *
 * Returns: True or false if there is a path that exists
 *
 * Updates: Parent params of nodes for backtracing and MinResCap
 * for updating capacity along arcs in residual graph
 */
public boolean findPathFromSourceToSink(Node s, Node t) { // BFS
    int minResCap = Integer.MAX_VALUE; // stores
    for (Node node : this.nodes) {
        node.viewed = false;
        node.parent = null;
    }

    LinkedList<Node> queue = new LinkedList<Node>();
    s.viewed = true;
    s.parent = null;
    queue.add(s);

    while (queue.size() != 0) {
        Node cur = queue.poll();
        // System.out.println("searched: " + cur.name);
        for (Arc arc : arcs) {
            if (arc.n1.equals(cur) && !arc.n2.viewed && arc.u > 0) {
                if (arc.u < minResCap) {
                    minResCap = arc.u;
                    // System.out.println("updated minrescap to " +
                        minResCap + " from arc " +
                    // arc.name);
                }
                arc.n2.parent = arc.n1;

                if (arc.n2.equals(t)) {
                    // System.out.println("final minrescap: " +
                        minResCap);
                    arc.n2.minResCap = minResCap;
                    return true;
                }

                queue.add(arc.n2);
                arc.n2.viewed = true;
            }
        }
    }
    return false;
}
/******** helper function: FORD-FULKERSON ********/
/*
 * Returns: void
 */
public void modifiedFordFulkerson(Node s, Node t) {
```

```
    while (this.findPathFromSourceToSink(s, t)) {
        for (Node cur = t; !cur.equals(s); cur = cur.parent) {
            for (Arc rarc : this.arcs) {
                if (rarc.n1.equals(cur) && rarc.n2.equals(cur.parent))
                    { // add min residual capacity to reverse arc
                    rarc.u += t.minResCap;
                } else if (rarc.n2.equals(cur) &&
                    rarc.n1.equals(cur.parent)) { // subtract
                    residual capacity from
                                                                    //
                                                                        forward
                                                                        arc

                    rarc.u -= t.minResCap;
                }
            }
        }
    }
}

/******** Generate feasible flow ********/
/*
 * Returns: Boolean if a feasible flow has been detected
 */
public boolean generateFeasibleFlow() {
    Graph rFordFulkerson = new Graph();
    rFordFulkerson.initResidualGraph(this);

    int nameSource = -1;
    int nameSink = -2;
    Node s = new Node(nameSource, 0);
    rFordFulkerson.nodes.add(s);
    Node t = new Node(nameSink, 0);
    rFordFulkerson.nodes.add(t);

    for (Node node : rFordFulkerson.nodes) {
        if (node.supply > 0) {
            Arc sourceArc = new Arc("s" + node.name, s, node,
                node.supply, 0); // capacity along source arc is equal
                                                                //
                                                                    to
                                                                    supply
                                                                    of
                                                                    the
                                                                    source
                                                                    node
            rFordFulkerson.arcs.add(sourceArc);
            Arc rSourceArc = new Arc("rs" + node.name, node, s, 0, 0);
            rFordFulkerson.arcs.add(rSourceArc);
        } else if (node.supply < 0) {
```

```
            Arc sinkArc = new Arc("t" + node.name, node, t, -1 *
                node.supply, 0); // capcity along sink arc is equal
                                                                    //
                                                                        to
                                                                        the
                                                                        opposite
                                                                        of
                                                                        the
                                                                        supply
                                                                    //
                                                                        of
                                                                        the
                                                                        sink
                                                                        node
            rFordFulkerson.arcs.add(sinkArc);
            Arc rSinkArc = new Arc("rs" + node.name, t, node, 0, 0);
            rFordFulkerson.arcs.add(rSinkArc);
        }
    }

    rFordFulkerson.modifiedFordFulkerson(s, t);

    for (Arc rarc : rFordFulkerson.arcs) { // There is a feasible
        flow if and only if capacity is maximized
                                        // (residual
        // capacity = 0) from all edges flowing out of s and into t
        if (rarc.n1.equals(s)) {
            if (rarc.u != 0) {
                return false;
            }
        } else if (rarc.n2.equals(t)) {
            if (rarc.u != 0) {
                return false;
            }
        }
    }

    for (Arc arc : this.arcs) { // update flow on arcs of original
        graph
        for (Arc rarc : rFordFulkerson.arcs) {
            if (arc.n1.equals(rarc.n1) && arc.n2.equals(rarc.n2)) {
                arc.x = arc.u - rarc.u;
            }
        }
    }
    return true;
}

/******** helper function: Modified Bellman-Ford Algorithm ********/
/*
```

```
 * Returns: Boolean if a a negative cycle has been detected
 */
public boolean modifiedBellmanFordNegCycleDetect() {
    // 1) establish parent and distance variables with initial
        conditions
    for (Node node : nodes) {
        node.parent = null;
        node.inNegCycle = false;
        node.dist = Integer.MAX_VALUE;
        // set distance for source nodes to 0, distance to itself is 0
        if (node.supply > 0) {
            node.dist = 0;
        }
    }

    // 2) Relax each index in our node set by updating shortest
        distance to each
    // node |nodes| - 1 times
    for (int i = 0; i < nodes.size(); i++) {
        for (Arc arc : arcs) {
            if ((arc.n2.dist > arc.n1.dist + arc.c) && (arc.n1.dist
                != Integer.MAX_VALUE)) {
                arc.n2.dist = arc.n1.dist + arc.c;
                arc.n2.parent = arc.n1;
            }
        }
    }
    List<Node> cycle = new ArrayList<>();
    Node foundCycle = null;
    // 3) Repeat to detect any negative cycles with arcs with
        positive residual
    // capacity, return 0 if detected (step 2 should
    // be optimal, so if there is another shorter path, there must
        be a negative
    // cycle)
    for (Arc arc : arcs) {
        if ((arc.n2.dist > arc.n1.dist + arc.c) && (arc.n1.dist !=
            Integer.MAX_VALUE) && (arc.u > 0)) {
            foundCycle = arc.n2;
            break;
        }
    }
    // 4) if no negative cycle detected, return false
    if (foundCycle == null) {
        return false;
    }

    //not necessary, used for testing
    Node startNode = foundCycle;
    Node prevNode = null;
```

```
    while (true) {
        prevNode = startNode.parent;
        if (cycle.contains(prevNode)) {
            break;
        }
        cycle.add(prevNode);
        prevNode.inNegCycle = true;
        startNode = prevNode;
    }

    int i = cycle.indexOf(prevNode);
    for (int j = 0; j < i; j++) {
        cycle.remove(j);
    }

    return true;
}

/******** NEGATIVE CYCLE CANCELING ALGORITHM ********/
/*
 * Returns: Boolean, true if a a negative cycle has been detected
     and the updates have been performed
 *                 false if no negative cycle has been detected,
     optimal solution has been found
 */
public boolean negativeCycleCanceling() {
    // 1) generate residual graph
    Graph rNCC = new Graph();
    rNCC.initResidualGraph(this);

    // 2) detect negative cycles (function is above)
    if (rNCC.modifiedBellmanFordNegCycleDetect()) {
        Set<Arc> negCycle = new HashSet<>();
        int uc = Integer.MAX_VALUE;

        // 3) search arcs for arcs in negative cycle based on parent
            vals and store minimum residual capacity
        for (Arc rarc : rNCC.arcs) {
            if (rarc.n1.inNegCycle && rarc.n2.inNegCycle &&
                rarc.n2.parent.equals(rarc.n1)) {
               negCycle.add(rarc);
               if (rarc.u < uc) {
                   uc = rarc.u;
               }

            }
        }
        //if the residual capacity is negative, do nothing and
            iterate again (test for all positive residual capacity)
        if (uc < 0) {
```

```
            return true;
        }

        for (Arc negCycleArc : negCycle) { // update residual
            capacity with minimum residual capacity along arcs in
            neg cycle
            for (Arc rarc : rNCC.arcs) {
                if (negCycleArc.n1.equals(rarc.n1) &&
                    negCycleArc.n2.equals(rarc.n2)) {
                    rarc.u -= uc;
                } else if (negCycleArc.n1.equals(rarc.n2) &&
                    negCycleArc.n2.equals(rarc.n1)) {
                    rarc.u += uc;
                }
            }
        }
        for (Arc arc : this.arcs) { // update flow on arcs of
            original graph
            for (Arc rarc : rNCC.arcs) {
                if (arc.n1.equals(rarc.n1) && arc.n2.equals(rarc.n2)) {
                    arc.x = arc.u - rarc.u;
                }
            }
        }
        return true;
    }
    //return false to exit if no negative cycle has been detected
    return false;

}

public static void main(String[] args) throws Exception {
    Graph g = new Graph();
    g.initGraph();

    if (!g.generateFeasibleFlow()) {
        System.out.println("There is no feasible flow in the selected
            graph");
        System.exit(0);
    }
    System.out.println("Feasible flow detected, running negative
        cycle cancelling on the following feasible flow:");
    for (Arc arc : g.arcs) {
        System.out.println(arc.name + ": " + arc.x);
    }
    int iter = 0;
    while (g.negativeCycleCanceling()) {
        iter++;
        System.out.println(iter);
        System.out.println("updated flow: ");
```

```
            for (Arc arc : g.arcs) {
                System.out.println(arc.name + ": " + arc.x);
            }
        }
        System.out.println("Final minimum cost flow after negative cycle
            canceling is: ");
        int sum = 0;
        for (Arc arc : g.arcs) {
            System.out.println(arc.name + "| flow: " + arc.x + ", cost: "
                + (arc.c * arc.x));
            sum += (arc.c * arc.x);
        }
        System.out.println("Final Minimum cost: " + sum);
    }
}
```