Network Flows II

The Ford-Fulkerson algorithm discussed in the last class takes time $O(mF)$, where F is the value of the maximum flow, when all capacities are integral. This is fine if all edge capacities are small, but if they are large numbers then this could even be exponentially large in the description size of the problem. In this lecture we examine some improvements to the Ford-Fulkerson algorithm that produce much better (polynomial) running times regardless of the value of the max flow or capacities.

We will then consider a generalization of max flow called *minimum-cost* flows, where edges have costs as well as capacities, and the goal is to find flows with low cost. This problem has a lot of nice properties, and is also highly practical since it generalizes the max flow problem.

Objectives of this lecture

In this lecture, we will:

- See an algorithm for max flow with polynomial running time (Edmonds-Karp)
- Define and motivate the *minimum-cost flow* problem
- Derive and analyze some algorithms for minimum-cost flows

Recommended study resources

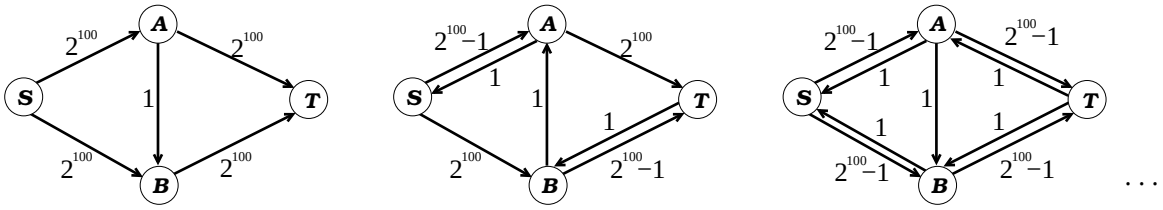
- CLRS, *Introduction to Algorithms*, Chapter 26, Maximum Flow
- DPV, *Algorithms*, Chapter 7.2, Flows in Networks
- Erikson, *Algorithms*, Chapter 10, Maximum Flows & Minimum Cuts

1 Network flow recap

Recall that in the maximum flow proble, we are given a directed graph G, a source s, and a sink t. Each edge (u,v) has some capacity $c(u,v)$, and our goal is to find the maximum flow possible from s to t. Last time we looked at the Ford-Fulkerson algorithm, which we used to prove the min-cut max-flow theorem, as well as the integrality theorem for flows. The Ford-Fulkerson algorithm is a greedy algorithm: we find a path from s to t of positive capacity and we push as much flow as we can on it (saturating at least one edge on the path). We then describe the capacities left over in a "residual graph", which accounts for remaining capacity as well as

the ability to redirect existing flow (and hence "undo" bad previous decisions) and repeat the process, continuing until there are no more paths of positive residual capacity left between s and t. We then proved that this in fact finds the maximum flow.

Assuming capacities are integers, the basic Ford-Fulkerson algorithm could make up to F iterations, where F is the value of the maximum flow. Each iteration takes $O(m)$ time to find a path using DFS or BFS and to compute the residual graph. (We assume that every vertex in the graph is reachable from s, so $m \geq n-1$.) So, the overall total time is $O(mF)$. This is fine if F is small, like in the case of bipartite matching (where $F \leq n$). However, it's not good if capacities are large and F could be very large. Here's an example that could make the algorithm take a very long time. If the algorithm selects the augmenting paths $s \to A \to B \to t$, then $s \to B \to A \to t$, repeating..., then each iteration only adds one unit of flow, but the max flow is 2^{101}, so the algorithm will take 2^{101} iterations. If the algorithm selected the augmenting paths $s \to A \to t$ then $s \to B \to t$, it would be complete in just two iterations! So the question on our minds today is can we find an algorithm that provably requires only polynomially many iterations?



2 Shortest Augmenting Paths Algorithm (Edmonds-Karp)

There are several strategies for selecting better augmenting paths than arbitrary ones. Here's one that is quite simple and has a provable polynomial runtime. Its called the Shortest Augmenting Paths algorithm, or the Edmonds-Karp algorithm. The name of the algorithm might give away a slight hint of what it does.

Algorithm: Shortest Augmenting Paths (Edmonds-Karp)

Edmonds-Karp implements Ford-Fulkerson by selecting the *shortest* augmenting path each iteration.

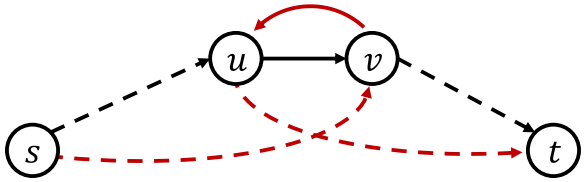
Unsurprisingly, the Shortest Augmenting Paths (Edmonds-Karp) algorithm works by always picking the *shortest* path in the residual graph (the one with the fewest number of edges). By example, we can see that this would in fact find the max flow in the graph above in just two iterations, but what can we say in general? In fact, the claim is that by picking the shortest paths, the algorithm makes at most mn iterations. So, the running time is $O(nm^2)$ since we can use BFS in each iteration. The proof is pretty neat too.

Theorem: Runtime of Edmonds-Karp

The Shortest Augmenting Paths algorithm (Edmonds-Karp) makes at most mn iterations.

Proof. Let d be the distance from s to t in the current residual graph. We'll prove the result by showing:

Claim (a): d never decreases Consider one iteration of the algorithm. Before adding flow to the augmenting path, every vertex v in G has some distance d_v from the source vertex s *in the residual graph.* Suppose the augmenting path consists of the vertices $v_1, v_2, \ldots v_k$. What can we say about the distances of the vertices? Since the path is by definition a *shortest path*, it must be true that $d_{v_i} = d_{v_{i-1}} + 1$, that is, every vertex is one further from s. Now perform the augmentation and consider what changes in the residual graph. Some of the edges (at least one) become *saturated*, which means that the flow on the edge reaches its capacity. When this happens, that edge will be removed from the residual graph. But another edge might appear in the residual graph! Specifically, when $e = (u, v)$ goes from zero to nonzero flow, $e' = (v, u)$ may appear in the residual graph as a back edge (if it doesn't exist already). Can this lower the distance of any vertex? No, $d_v = d_u + 1$, so adding an edge from v to u can't make a shorter path from s to t.



Therefore, since the distance to any vertex can not decrease, d can not decrease.

Claim (b): every m iterations, d has to increase by at least 1 Each iteration saturates (fills to capacity) at least one edge. Once an edge is saturated it can not be used because it will not appear in the residual graph. For the edge to become usable again, it must be the case that its back edge in the residual graph is used, which means that the back edge needs to appear on the shortest path. However, if $d_v = d_u + 1$, then it is not possible for the back edge (v, u) to be on a shortest path, so this can *only* occur if d increases. Since there are m edges, d must increase by at least one every m iterations.

Since the distance between s and t can increase at most n times, in total we have at most nm iterations. □

This shows that the running time of this algorithm is $O(nm^2)$. Note that this is true for **any** capacities, including large ones and non-integer ones. So we really have a polynomial-time algorithm for maximum flow!

3 Minimum-Cost Flows

We talked about the problem of assigning groups to time-slots where each group had a list of acceptable versus unacceptable slots. This was the *bipartite matching* problem. A natural generalization is to ask: what about preferences? E.g, maybe group A prefers slot 1 so it costs

only $1 to match to there, their second choice is slot 2 so it costs us $2 to match the group here, and it can't make slot 3 so it costs $infinity to match the group to there. And, so on with the other groups. Then we could ask for the *minimum-cost* perfect matching. This is a perfect matching that, out of all perfect matchings, has the least total cost.

The generalization of this problem to flows is called the *minimum-cost flow* problem.

Problem: Minimum-cost flow

We are given a directed graph G where each edge has a *cost*, $\$(e)$, and a capacity, $c(e)$. As before, an *s-t flow* is an assignment of values to edges that satisfy the capacity and conservation constraints, and the *value* of a flow is the net outflow of the source s. The **cost** of a flow is then defined as the sum over all edges, of the cost per unit of flow:

$$\text{cost}(f) = \sum_{e \in E} \$(e) f(e).$$

Remark: Cost is per unit of flow

Note **importantly** that the cost is charged *per unit* of flow on each edge. That is, we are not paying a cost to "activate" an edge and then send as much flow through it as we like. This similar problem turns out to be NP-Hard.

The goal of the minimum-cost flow problem is to find feasible flows of minimum possible cost. There are a few different variants of the problem.

- We will consider the **minimum-cost maximum flow** problem, where we seek to find the minimum-cost flow out of all possible maximum flows.
- An alternative formulation is to try to find the minimum-cost flow of value k for some given parameter k, rather than a maximum flow.

These problems can be reduced to each other so they are all equivalent. There are also many other variants of the problem, but we won't consider those for now. Formally, the min-cost max flow problem is defined as follows. Our goal is to find, out of all possible maximum flows, the one with the least total cost. What should we assume about our costs?

- In this problem, we are going to allow *negative costs*. You can think of these as rewards or benefits on edges, so that instead of paying to send flow across an edge, you get paid for it.
- What about negative-cost cycles? It turns out that minimum-cost flows are still perfectly well defined in the presence of negative cycles, so we can allow them, too! Some algorithms for minimum-cost flows however don't work when there are negative cycles, but some do. We will be explicit about which ones do and do not.

Min-cost max flow is more general than plain max flow so it can model more things. For example, it can model the min-cost matching problem described above.

3.1 The residual graph for minimum-cost flows

Let's try to re-use the tools that we used to solve maximum flows for minimum-cost flows. Remember that our key most important tool there was the *residual graph*, which we recall has edges with capacities

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E. \end{cases}$$

We will re-use the residual graph to attack minimum-cost flows. We need to generalize it, though, because our current definition of the residual graph has no ideas about the costs of the edges. For forward edges e in the residual graph (i.e., those corresponding to $e = (u,v) \in E$), the intuitive value to use for their cost is $\$(e)$, the cost of sending more flow along that edge. What about the reverse edges, though? That's less obvious. Let's think about it, when we send flow along a reverse in the residual graph, we are removing or *redirecting* the flow somewhere else. Since it cost us $\$(e)$ per unit to send flow down that edge initially, when we remove flow on an edge, we can think of getting a *refund* of $\$(e)$ per unit of flow – we get back the money that we originally paid to put flow on that edge. Therefore, the right choice of cost for a reverse edge in the residual graph is $-\$(e)$, the negative of the cost of the corresponding forward edge!

Definition: The residual graph for minimum-cost flows

The residual graph G_f for a minimum-cost flow problem has the same capacities as the original maximum flow problem, but now has costs of $\$(e)$ on a forward edge, and $-\$(e)$ on a back edge. That is, suppose we denote by $\overleftarrow{e}$, the corresponding reverse edge of e in the residual graph. We have

$$c_f(e) = c(e) - f(e), \qquad \$_f(e) = \$(e),$$
$$c_f(\overleftarrow{e}) = f(e) \qquad \$_f(\overleftarrow{e}) = -\$(e)$$

The definitions on the left are the same as regular max flow.

4 An augmenting path algorithm for minimum-cost flows

Now that we have defined a suitable residual graph for minimum-cost flows, we can build an algorithm based on augmenting paths in the same spirit as Ford-Fulkerson. If we just try to pick arbitrary augmenting paths, then it is unlikely that we will find the one of minimum cost, so how should we select our paths in such a way that the costs are accounted for? The most intuitive idea would be to select the *cheapest augmenting path*, i.e., the shortest augmenting path with respect to the costs. This is not to be confused with the shortest augmenting path algorithm that selects the path with the fewest edges (i.e., the Edmonds-Karp algorithm).

Since the edges are weighted by cost, a breadth-first search won't work anymore. How about our favorite shortest paths algorithm, Dijkstra's? Well, that won't quite work since the graph is going to have negative edge costs (note that even if the input graph does not have any negative

costs, the residual graph will, so Dijkstra's will not work here). So, let's use our next-favorite shortest paths algorithm that is capable of handling negative edges: Bellman-Ford! It is important to note here that since the algorithm makes use of shortest path computations, if there is a negative-cost cycle, this algorithm will not work.

Algorithm: Cheapest augmenting paths

Implement Ford-Fulkerson by selecting the *cheapest* augmenting path with respect to residual costs.

While there exists any augmenting path in the residual graph G_f, find the augmenting path with the cheapest cost and augment as much flow as possible along that path. Since this is just a special case of Ford-Fulkerson, the fact this algorithm finds a maximum flow is immediate, right? Well, not quite. Remember that since shortest paths only exist if there is no negative-cost cycle, we need to prove that our algorithm never creates one after performing an augmentation, or the next iteration's shortest path computation will fail.

This result will be a direct corollary of a cool lemma. This lemma should seem familiar and intuitive since it is really just the weighted version of the lemma used to prove the correctness of the Edmonds-Karp algorithm last time!

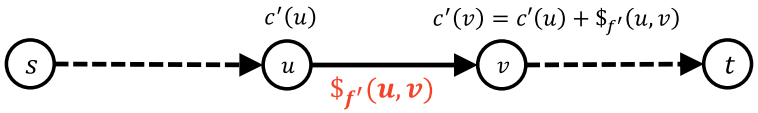
Lemma 1: Cheapest path does not decrease

Consider a flow network with costs G and a flow f such that G_f contains no negative-cost cycles. Let f' denote a flow obtained by augmenting f with a cheapest augmenting path from G_f. Then, the cost of the cheapest s-t path in $G_{f'}$ is at least as large as the cheapest s-t path in G_f. (In other words, the cheapest path never gets cheaper!)

Proof. Let G be a flow network with costs and f be a flow such that G_f contains no negative-cost cycles. Let us denote by $c(v)$, the cost of the cheapest path in G_f from s to v for any v. Suppose we augment f with a cheapest augmenting path from G_f to obtain a new flow f'.

Suppose for the sake of contradiction that there exists a walk[1] from s to t whose cost is cheaper than $c(t)$. Let v be the earliest vertex such that the cost of walking from s to v along this walk is cheaper than $c(v)$, and denote this cost by $c'(v) < c(v)$. Then, let u be the vertex directly preceding v on the walk, and denote the cost of walking to that occurrence of u by $c'(u)$.

Since u precedes v, the cost of v is $c'(v) = c'(u) + \$_{f'}(u,v)$.

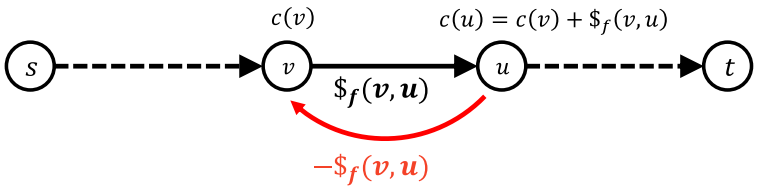


[1] A walk is a path that might contain repeated vertices. Technically we are required to consider walks instead of paths because hypothetically if a negative-cost cycle were to appear in $G_{f'}$, the resulting cheaper "paths" would contain repeated vertices.

Since v is the earliest such vertex on the path, we know that $c'(u) \geq c(u)$ (it might be greater, not equal, because an edge that was previously on the cheapest path got saturated), so

$$c'(v) \geq c(u) + \$_{f'}(u, v).$$

Now, if (u, v) had the same residual cost in G_f and $G_{f'}$ (i.e., $\$_{f'}(u, v) = \$_f(u, v)$), then $c'(v) \geq c(u) + \$_f(u, v) \geq c(v)$, which would contradict $c'(v) < c(v)$, so it must be that the residual cost of (u, v) changed. This means that the augmenting path must have used (v, u) and activated the corresponding back edge (u, v). Therefore $\$_{f'}(u, v) = -\$_f(v, u)$.



So, we have $c'(v) \geq c(u) - \$_f(v, u)$, however, in G_f note that since $c(u) = c(v) + \$_f(v, u)$, we also have $c(v) = c(u) - \$_f(v, u)$, which would imply that $c'(v) \geq c(v)$, a contradiction. □

Corollary 1: Maintenance of minimum-cost flows

If G_f contains no negative-cost cycles, then after augmenting with a cheapest augmenting path, it still contains no negative-cost cycles.

Proof. If there were a negative-cost cycle, then we could use that cycle to create paths of arbitrarily low cost, which would immediately imply that the cheapest path to t would be cheaper than before (or undefined). □

So, we have successfully proven that the algorithm will terminate, and since it is a special case of Ford-Fulkerson, we already know that it terminates with a maximum flow! We can also argue about the runtime using the same logic that we used for Ford-Fulkerson. At every iteration of the algorithm, at least one unit of flow is augmented, and every iteration has to run the Bellman-Ford algorithm which takes $O(nm)$ time. Therefore, the worst-case running time of cheapest augmenting paths is $O(nmF)$, where F is the value of the maximum flow, assuming that the capacities are integers.

What we still have to prove, however, is that this algorithm truly finds a *minimum-cost* flow.

5 An optimality criteria for minimum-cost flows

When studying maximum flows, our ingredients for proving that a flow was maximum were augmenting paths and the minimum cut. We'd like to find a similar tool that can be used to prove that a minimum-cost flow is optimal. First, let's be specific about what we mean by optimality.

Definition: Cost optimality of flows

We will say that a flow f is *cost optimal* if it is the cheapest of all possible flows *of the same value*.

That is to say we don't compare the costs of different flows if they have different values. Our goal is to find a tool to help us analyze the cost optimality of a flow. To do so, we're going to think about what kinds of operations we can do to modify a flow and change its cost *without* changing its value.

When finding maximum flows, our key tool was the *augmenting path*. If there existed an augmenting path in G_f, then by adding flow to it, we could transform a given flow into a more optimal one that was still feasible because adding flow to an s-t path preserved the flow conservation condition, and didn't violate the capacity constraint as long as we added an appropriate amount of flow. Therefore, the existence of an augmenting path proved that a flow was not maximum, and we were able to later prove that the lack of existence of an augmenting path was sufficient to conclude that a flow was maximum. We want to discover something similar for cost optimality of a flow. Augmenting paths were just one way to modify a flow while keeping it feasible. I claim that there is one other way that we can modify a flow while still preserving feasibility, but that also doesn't change its value. Instead of adding flow to an s-t path, what if we instead add flow to a *cycle* in the graph? Let's call this an *augmenting cycle*.

Since a cycle has an edge in and an edge out of every vertex, adding flow to a cycle in the residual graph preserves flow conservation, and hence feasibility. Furthermore, since the same amount of flow goes in and out of each vertex, the flow value is unaffected! However, adding flow to a cycle may in fact change the cost of the flow, which is what we wanted! Suppose the costs of the edges $e_1, e_2, ..., e_k$ on the cycle are $\$_1, \$_2, ..., \$_k$ for some cycle of length k. Then, adding Δ units of flow to the cycle will change the cost by

$$\sum_{i=1}^{k} \Delta \cdot \$_i = \Delta \cdot \sum_{i=1}^{k} \$_i.$$

Now, note that $\sum \$_i$ is just the weight of the cycle! So, we can say that if we add Δ units of flow to a cycle of weight W, then the cost of the flow changes by $\Delta \cdot W$. If $\Delta \cdot W$ is negative, then we have just shown that the cost of the flow *can be decreased*, so it wasn't optimal!

It turns out that this is the key ingredient for analyzing minimum-cost flows. Even better, we are getting a two-for-one deal because a lack of negative-cost cycles was what we already argued was required for our cheapest augmenting path algorithm to produce a maximum flow. It turns out that this same condition allows us to prove that the flow is cost optimal. We just need a few more concepts first, then we are done, I promise!

5.1 Differences of flows and circulations

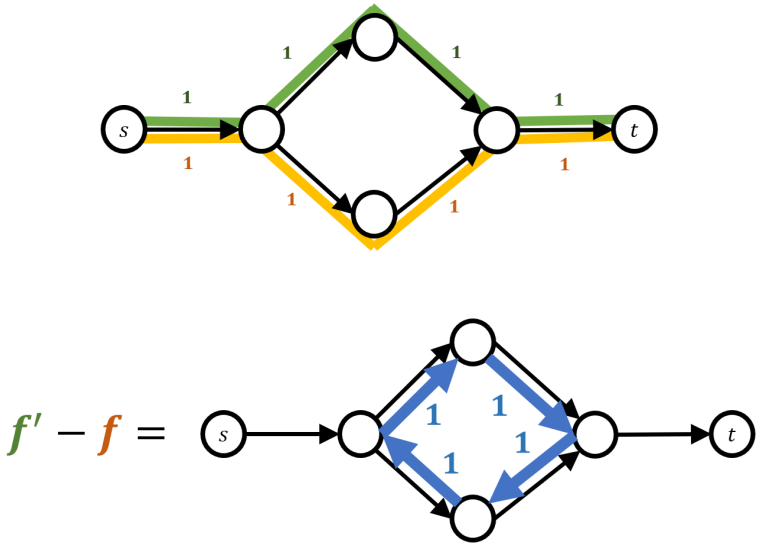
We just argued that the presence of a negative-cost cycle implies that we can find a flow of lesser cost by *adding* two flows together. Adding flows together is defined in the natural way you would think of. We add the values of the flow on each edge to get a new flow. One can show

from the definition that if a flow f is feasible in G and another flow f' is feasible in G_f, then $f + f'$ is feasible in G (where we define adding flow to the reverse edge as removing flow from the corresponding forward edge, just like in Ford-Fulkerson).

To complete the analysis of minimum cost flows, we will also need the slightly less intuitive notion of the *difference* between two flows. Given a flow f' and a flow f, what would we want $f' - f$ to mean? We could just take the edgewise difference between each of the flows, but this could sometimes result in a negative amount of flow which does not quite make sense. However, the following idea will make it make sense!

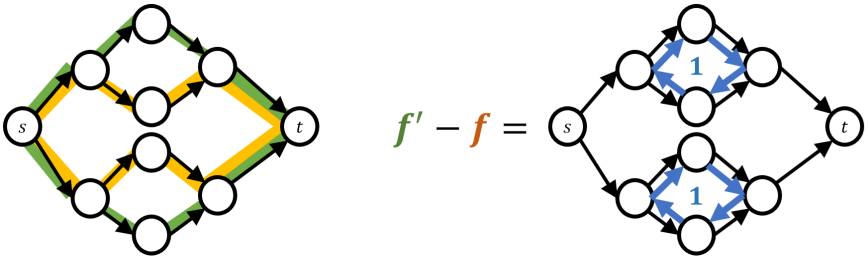
- if $f'(u,v) - f(u,v) \geq 0$, then we will say that (u,v) has $f'(u,v) - f(u,v)$ flow,
- if $f'(u,v) - f(u,v) < 0$, then we will say that (v,u) has a flow of $f(u,v) - f'(u,v)$.

In other words, a negative amount of flow will just be treated as a positive amount of flow going in the other direction! This is analogous to the way that reverse edges are treated by the Ford-Fulkerson algorithm; they remove flow in the other direction. Given two flows, what does this actually look like? Suppose we have two flows f' and f *of the same value*. In this example, the difference between the green flow and the yellow flow is that the flow on the edges adjacent to s and t cancel, and the flow around the bottom two edges *reverses*. This results in a *cycle* around the middle of the graph!



Now the claim is that this is what always happens: the difference between two flows of the same value is a collection of (possibly multiple) cycles. But how would we prove this, and how do we know that it is even a valid flow? Well, since both f' and f are feasible flows, they satisfy the flow conservation property, and hence their difference $f' - f$ also satisfies the flow conservation property (for example, if some vertex has 5 flow in and out in f' and 3 flow in and out in f, then their difference has 2 flow in and out). What is the value of this flow? By assumption, the values of f' and f are the same, hence their net flows out of s are the same. By taking the difference between the two, we can see that the net flow out of s in $f' - f$ is actually **zero**. So, we have

a flow of value zero, but it isn't the all-zero flow, so what does this look like? Well, every vertex *including the source and sink* have flow in equal to flow out, which means that the flow is indeed a *collection of cycles*. For this reason, such a flow is often called a *circulation*.



5.2 Proving the optimality criteria

Theorem 1: Cost optimality and negative cycles

A flow f is cost optimal if and only if there is no negative-cost cycle in G_f.

Proof. Suppose that there exists a negative cost cycle in the residual graph. Then by adding flow to this cycle, the value of the flow doesn't change, but the cost changes by $\Delta \cdot W$, where Δ is the amount of flow we can add, and W is the cost/weight of the cycle. Since W is negative, the cost decreases, and hence the flow f was not cost optimal.

Now suppose that a flow f is not cost optimal. We need to prove that there exists a negative-cost cycle in its residual graph G_f. This case is much trickier. Since f is not cost optimal, there exists some other flow f' of the same value but which has a cheaper cost. Let's now consider the difference flow $f'-f$. From the previous section, this is a *circulation*, i.e., a collection of cycles of flow. What is the cost of this circulation? Since costs are just linear (we sum the cost per flow along each edge), we get that

$$\text{cost}(f'-f) = \text{cost}(f') - \text{cost}(f),$$

and since we assumed that $\text{cost}(f') < \text{cost}(f)$, this must be a *negative cost*. Now we wish to show that $f'-f$ is *feasible in* G_f. We consider the two cases in the definition of $f'-f$:

- If $f'(e)-f(e) \geq 0$, then the forward edge e has flow $f'(e)-f(e) \leq c(e)-f(e)$ which is the definition of the residual capacity $c_f(e)$,
- if $f'(e)-f(e) < 0$, then the reverse edge $\vec{e}$ has flow $f(u,v)-f'(u,v) \leq f(u,v)$, which is the definition of the residual capacity of $c_f(\vec{e})$.

Therefore, $f'-f$ is a feasible flow in the residual graph! So, to conclude, we have a collection of cycles of flow whose total cost is negative, therefore at least one of those cycles must have a negative cost. Since $f'-f$ is feasible in the residual graph, this negative-cost cycle exists in the residual graph, which is what we wanted to prove. □

Corollary: Optimality of cheapest augmenting paths

Since we already argued in Corollary 1 that the cheapest augmenting path algorithm never creates a negative-cost cycle in the residual graph, this proves that the resulting flow is also cost optimal. Therefore we can conclude that the cheapest augmenting path algorithm successfully finds the minimum-cost maximum flow!