Maximum Flow and its Applications: Part 1

Saladi Rahul, Sept. 5th 2023

1 Introduction

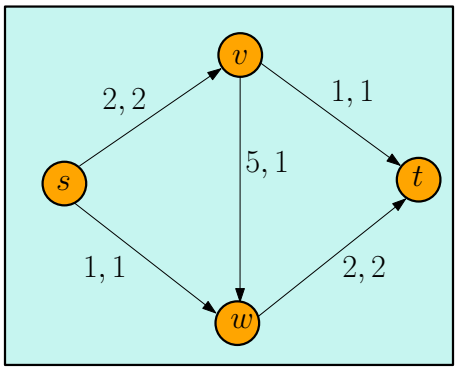

Figure 1: A flow of three units being sent from the source s to the sink t. On each edge, the first label represents the capacity of the edge and the second label represents the flow going through the edge.

Warm-up. In the above example, can a flow of four units be sent from the source to the sink? Suppose the capacity of the edge (s, w) is increased to 2. Now can a flow of four units be sent from the source to the sink?

Problem statement. The input is a directed graph $G = (V, E)$. Each edge $e \in E$ has a positive integer *capacity* c_e. There is a special *source* vertex s and a special *sink* vertex t. There are no incoming edges into s and no outgoing edges from t. A *flow* function $f : E \to \mathbb{R}_{\geq 0}$ in G should satisfy the following two properties:

- *Capacity constraints.* For each edge $e \in E$, we have $f_e \leq c_e$.
- *Conservation constraints.* In plain english, the total flow entering a node v is equal to the total flow leaving v. Formally, for a vertex $v \in V \setminus \{s, t\}$, let $\delta^+(v)$ and $\delta^-(v)$ be the set of incoming and outgoing edges, respectively. Then for all $v \in V \setminus \{s, t\}$,

$$\sum_{e \in \delta^+(v)} f_e = \sum_{e \in \delta^-(v)} f_e.$$

The *value* of a flow f is the total amount of flow going out of the source, i.e., $\sum_{e \in \delta^-(s)} f_e$. The goal of the maximum flow problem is to *maximize* the amount of flow going out of the source.

Exercise. Prove that the amount of flow going out of the source is equal to the amount of flow going into the sink. Intuitively, this is easy to see, but proving it formally will give a good warm-up before we dive deep into the problem. *Hint: For a given flow function f , define $excess_f(v) = \sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e$, for all $v \in V$; and then prove that $\sum_{v \in V} excess_f(v) = 0$.*

Importance of maximum flow? There are some good reasons for studying the maximum flow problem:

- We will establish the famous *max-flow min-cut theorem*. This will connect seemingly different objects: maximum flow in a graph with the minimum cut in a graph.
- For many problems studied in the basic algorithms course, the correctness of the algorithm follows from the description of the algorithm. This is true for many divide and conquer algorithms such as mergesort, and dynamic programming based algorithms. However, for the Ford-Fulkerson algorithm which we will study next the optimality is not obvious. The optimality is established via connection to the minimum cut of the graph (an object which does not show up in the algorithm)!
- If you have ever wondered why you were taught different "proof techniques" in your discrete mathematics course, then you will get your answer here: establishing the max-flow min-cut theorem and the running time of Edmonds-Karp algorithm will demonstrate the importance of coming up with carefully written proofs. The algorithms by themselves are quite simple to state.
- The maximum flow problem by itself has many direct applications: such as routing packets in a network, transporting oil in a network of pipes, transporting Parle G biscuits from the factory to a particular destination[1]. Also, there are many other fundamental problems in computer science which can be *reduced* to an instance of maximum flow problem. We will see two such applications: densest subgraph problem and CSK's elimination in IPL.

2 A naive greedy algorithm

Most of the times greedy algorithms do not work. However, sometimes they give insights which can be useful in designing our final algorithm. We will look at one such algorithm.

A greedy algorithm. Initially, set $f_e \leftarrow 0$ for all $e \in E$ in G. As long as s and t are connected, keep performing the following steps:

- Pick *any* $s-t$ path P in G using the standard BFS or DFS approach.
- What is the maximum flow which can be sent on P? The answer is Δ, where

$$\Delta = \min_{e \in P} (c_e - f_e).$$

- Set $f_e \leftarrow f_e + \Delta$, for all $e \in P$. For all edges e on P with $f_e = c_e$, remove them from G, since no more flow can be sent through such edges.

Return the flow $\{f_e\}_{e \in E}$.

[1] Chai lovers favourite combo in India. So, more the flow of Parle G, the better!

Does it work? It turns out that the above greedy algorithm might not return the maximum flow. The issue is that we are allowed to pick *any* $s-t$ path in G. Consider Figure 2.

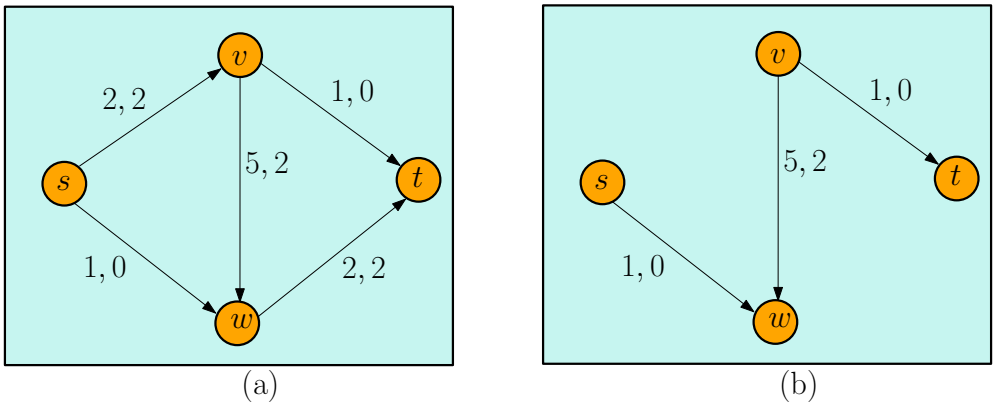

Figure 2: (a) If the initial $s-t$ path we choose is $s-v-w-t$, then a flow of two is sent. (b) After this edges (s, v) and (w, t) cannot send any more flow and hence, there is no path left from s to t.

3 The residual graph: undo our sins

The issue with the previous greedy algorithm is that once it makes a mistake, then it does not give itself a chance to "undo" those mistakes. Going back to Figure 2, in the maximum flow we would send only one unit of flow on the edge (v, w), whereas the greedy algorithm sent two units of flow on it. Therefore, it would be nice if we send some flow back on the edge (v, w) in the "reverse" direction. Specifically, if we could send one unit of flow on the edge (s, w), then route the unit flow in the "reverse" edge (w, v), and finally route the unit flow along the edge (v, t) then we will be able to get a maximum flow of three units. To enable such a feature, we will now define a *residual graph*.

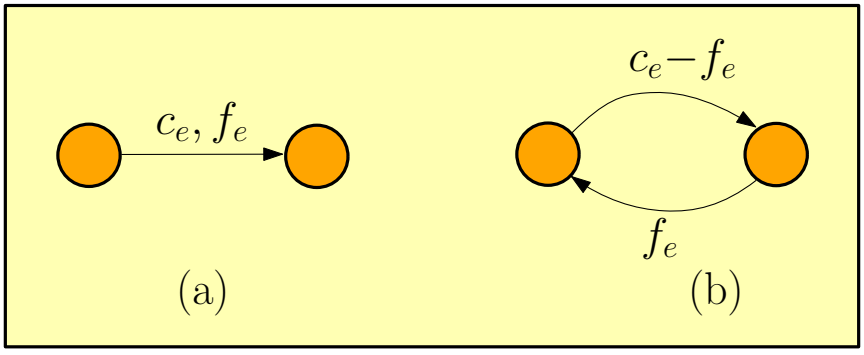

Figure 3: (a) An edge e in the original graph. (b) The forward edge and the reverse edge corresponding to e in the residual graph.

Given a graph $G = (V, E)$ and a flow function f, its residual graph $G_f = (V_f, E_f)$ is defined such that $V_f = V$ and for each edge $e = (u, v) \in E$, we add the following two edges to E_f: a *forward edge* (u, v) with residual capacity $c_e - f_e$ and a *reverse edge* (v, u) with residual capacity f_e.

See Figure 3. The capacity of the forward edge denotes the amount of flow which can still be sent through the edge e without violating the capacity constraint in the original graph G. The capacity of the reverse edge denotes how much flow can be *reduced* on the edge e in the original graph G while ensuring the flow is non-negative on the edge.

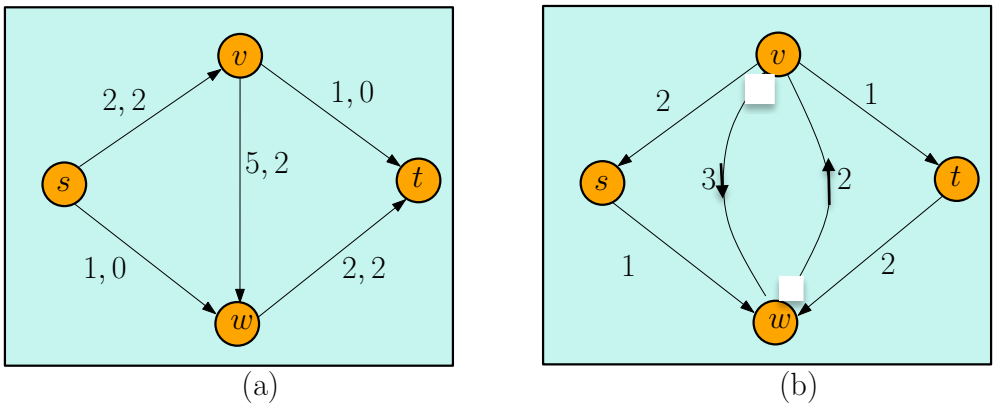

Figure 4: (a) Original graph G. The s–t path chosen is s–v–w–t and a flow of two is sent. (b) The corresponding residual graph G_f. The edges with capacity zero have been omitted.

4 Ford-Fulkerson algorithm

The Ford-Fulkerson algorithm can be summarized in one line as follows:

"Run the naive greedy algorithm on the residual graph."

Lets dive into the formal details. Initially, set $f_e \leftarrow 0$ for all $e \in E$ in G. As long as there is an s–t path P in G_f with all edges having greater than zero capacities:

- What is the maximum flow which can be sent on P? The answer is Δ, where

$$\Delta = \min_{e \in P} (\text{residual capacity of } e \text{ in } G_f).$$

  For any $e \in P$, let e' be the corresponding edge in G. Now the flow value is increased by Δ in G as follows:

- For all forward edges e in P, increase $f_{e'}$ by Δ. For all reverse edges e in P, decrease $f_{e'}$ by Δ.

Return the flow $\{f_e\}_{e \in E}$.

Applying Ford-Fulkerson algorithm on the residual graph in Figure 4(b) will give one unit flow on the path s–w–v–t. As a result the maximum flow of three units is achieved. So, there seems to some hope by using residual graphs!

5 Feasibility and termination

The Ford-Fulkerson algorithm is run on the residual graph. Therefore, one will have to verify if the capacity constraints and conservation constraints are not being violated in the original graph.

Lemma 1. *Throughout the algorithm, the capacity constraints and the conservation constraints are not violated in the original graph.*

Proof. By design of the residual graph, the capacity constraints are taken care of. Can you see that? For the conservation constraint, there are four cases to handle. Consider any any node v on the path P. Let e and e' be the edges on P such that e is the incoming edge into v and e' is the outgoing edge from v.

- If both e and e' are forward edges, then in the original graph, the flow into v and the flow out of v both increase by Δ.
- If both e and e' are reverse edges, then in the original graph the flow into v and the flow out of v both decrease by Δ.
- If e is a forward edge and e' is a backward edge, then the flow into v increases by $(\Delta - \Delta) = 0$.
- If e is a backward edge and e' is a forward edge: *by now you know how it works . . .*.

□

Lemma 2. *The capacity of each edge in the residual graph is integral. Also, the parameter Δ is always integral.*

Proof. All the capacities in the original graph are integral. Therefore, in the first iteration of the algorithm Δ will be integral, and as such the capacities in the residual graph will also be integral. By induction, assume these two statements are true at the end of the i-th iteration. In the $(i+1)$-th iteration, since all the edges in P have integral capacities, Δ will be integral, and hence, the new capacities in the residual graph will also be integral. □

Since $\Delta \geq 1$ in each iteration and since there is only a finite amount of flow which can leave the source, the algorithm will indeed terminate.