The Max-Flow Min-Cut Theorem

We now prove that the Ford-Fulkson algorithm is correct, i.e., when the algorithm terminates, the returned flow is a maximum-flow. Let f^* be the flow when the FF terminates. Note that the residual graph G_{f^*} w.r.t. f^* does not have any s-t path (or equivalently, s cannot reach t in G_{f^*}), because otherwise the FF algorithm will not terminate. Now we define a cut: let S^* be the set of vertices in G_{f^*} that can be reached from s, formally $S^* := \{v \in V \mid s \text{ can reach } v \text{ in } G_{f^*}\}$, and define $T^* = V \setminus S^*$. Clearly, $s \in S^*$. Since s cannot reach t in G_{f^*}, we know that $t \in T^*$. Hence (S^*, T^*) is an s-t cut. For example, in Figure 4, $S^* = \{s, b\}$ and $T^* = \{a, t\}$.

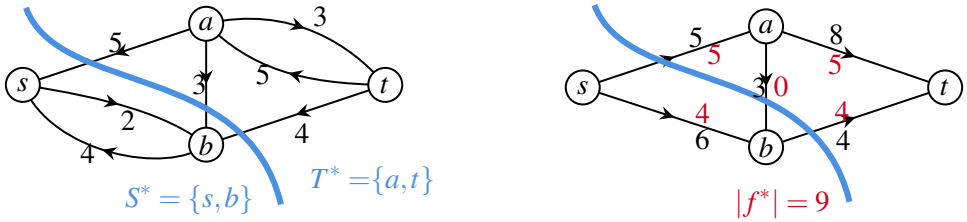


Figure 1: Illustration the proof of the max-flow min-cut theorem. Left: the residual graph G_{f^*} wrt the flow f^* when FF terminates (see Figure 4 of Lecture 19). The cut (S^*, T^*) is constructed by collecting vertices reachable from s in G_{f^*} as S^*, and $T^* = V \setminus S^*$. Right: the same s-t cut (S^*, T^*) drawn on the (original) graph G, where one can see that for every edge $e \in E(S^*, T^*)$, $f^*(e) = c(e)$, and for every edge $e \in E(T^*, S^*)$, $f^*(e) = 0$.

We now show the central result: $|f^*| = c(S^*, T^*)$. Recall the Fact 2 in Lecture 18: the value of any flow, here we focus on f^*, can be calculated with any s-t cut, and here we will use the particular cut (S^*, T^*). Specifically, we can write $|f^*| = \sum_{e \in E(S^*, T^*)} f^*(e) - \sum_{e \in E(T^*, S^*)} f^*(e)$.

In fact, we have $f^*(e) = c(e)$ for every $e \in E(S^*, T^*)$. See Figure 1. Why? Suppose conversely that $f^*(e) < c(e)$. Assume that $e = (u, v)$; the fact that $e = (u, v) \in E(S^*, T^*)$ means that $u \in S^*$ and $v \in T^*$. According to the way the residual graph is constructed, in G_{f^*}, there will be a forward edge (u, v). Consequently, s can reach v in G_{f^*}, as s can reach u (because $u \in S^*$) and u can reach v following this forward edge. This contradicts to the fact that $v \in T^*$.

Also, we can show that $f^*(e) = 0$ for every $e \in E(T^*, S^*)$. Why? Suppose conversely that $f^*(e) > 0$. Assume that $e = (u, v)$; the fact that $e = (u, v) \in E(T^*, S^*)$ means that $u \in T^*$ and $v \in S^*$. Then in the residual graph G_{f^*}, there will be a backward edge (v, u). Consequently, s can reach u in G_{f^*}, as s can reach v (because $v \in S^*$) and v can reach u following this backward edge. This contradicts to the fact that $u \in T^*$.

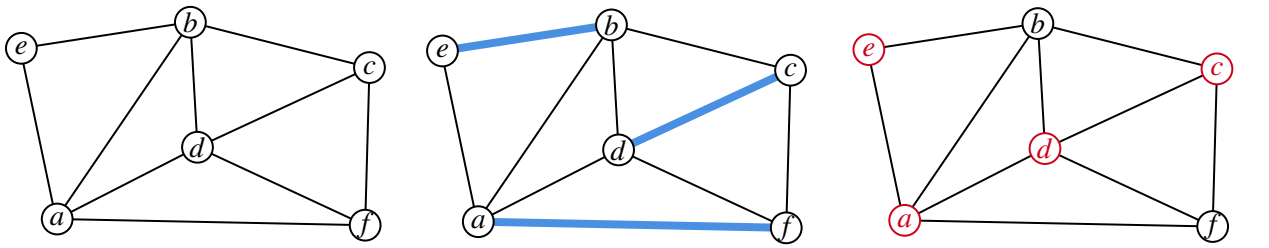
Combining all above, we have $|f^*| = \sum_{e \in E(S^*, T^*)} f^*(e) - \sum_{e \in E(T^*, S^*)} f^*(e) = \sum_{e \in E(S^*, T^*)} c(e) - 0 = c(S^*, T^*)$.

This proves that f^* is a maximum-flow and (S^*, T^*) is a minimum s-t cut, i.e., the FF algorithm is optimal, and a minimum-cut can be constructed from the residual graph. It also completes the connection between the maximum-flow problem and the minimum-cut problem: for any network, the value of the maximum-flow always equal to the capacity of the minimum-cut. This is called the max-flow min-cut theorem.

We analyze the time complexity of the Ford-Fulkson Algorithm. If we assume that all capacities are integers, we have $x(p) \geq 1$ and therefore the value of the new flow will be increased by at least 1 in each iteration. Hence, the number of iterations is upper-bounded by $|f^*|$. Each iteration includes building G_f, finding an s-t path p in G_f (with BFS/DFS), and augmenting p), which takes $O(|V| + |E|)$ time. Therefore, the FF algorithm runs in $O(|f^*| \cdot (|V| + |E|))$ time.

Maximum-Matching and Minimum-Vertex Cover

Let $G = (V, E)$ be an undirected graph. A *matching* $M \subset E$ is a subset of edges of G that does not have common vertices. A matching is also called an *independent edge set*. See Figure 2 for an example. Given an undirected graph G, the so-called *maximum-matching problem* seeks a matching M of G such that the number of edges in the matching, i.e., $|M|$, is maximized.



$G = (V, E)$ matching $M = \{(e,b),(d,c),(a,f)\}$ vertex cover $V_1 = \{e,a,d,c\}$

Figure 2: A matching M and a vertex cover V_1 of an undirected graph G. Note that M is also a *maximum matching* of G and V_1 is also a minimum vertex cover of G.

A *vertex cover* of an undirected graph $G = (V, E)$ is defined as a subset of vertices $V_1 \subset V$ that *covers* all edges, i.e., for every edge $(u, v) \in E$ either $u \in V_1$ or $v \in V_1$ or both. Given an undirected graph G, the so-called *minimum-vertex cover problem* seeks a vertex cover V_1 of G such that the number of verticesd in it, i.e., $|V_1|$, is minimized.

There exists a connection between matchings and vertex covers of an undirected graph, stated below.

Claim 1. Let $G = (V, E)$ be an undirected graph. Let M be any matching of G and let V_1 be any vertex cover of G. Then we have $|M| \leq |V_1|$.

Proof. Since V_1 is a vertex cover, it covers all edges of G, in particular, all edges of M. Since M is a matching, all edges in M does not share vertices, so it requires $|M|$ vertices to cover edges in M. Combined, we have $|M| \leq |V_1|$. □

So, the size of any vertex cover is an upper bound of the size of any matching. We again can virsualize such relationship with Figure 3. Clearly, if we can find a matching M and a vertex cover V_1 (of the same undirected graph G) satisfying that $|M| = |V_1|$, then M must be a maximum matching and V_1 must be a minimum vertex cover.

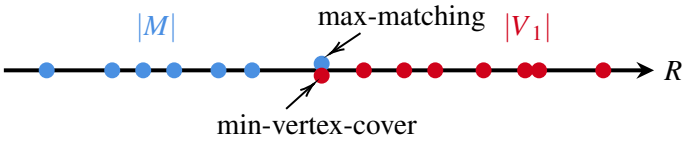


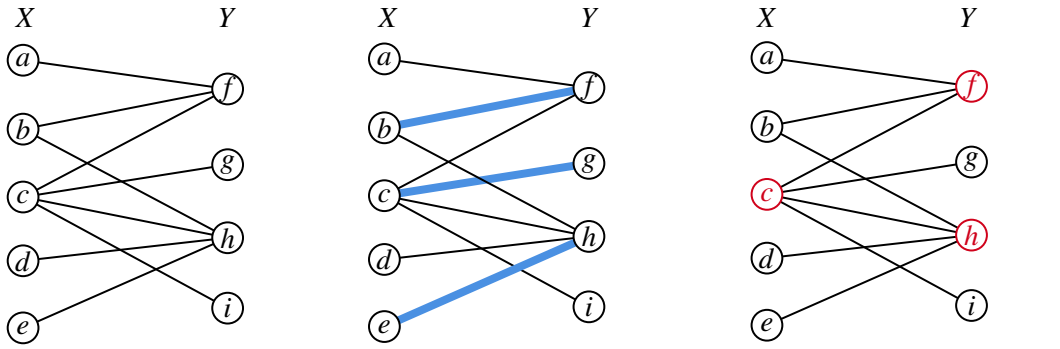
Figure 3: Each blue point represents the size of a matching and each red point represents the size of a vertex cover, of the same undirected graph. Do they always meet?

Does for *every* undirected graph there always exists a matching M and a vertex cover V_1 that satisfies $|M| = |V_1|$? The answer is no, and Figure 2 gives such an example. A more informative example is a triangle: a graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{(a, b), (b, c), (c, a)\}$. In a triangle, the maximum matching includes one edge, but to cover all 3 edges we need 2 vertices. In fact, an undirected graph consists of

an *odd-length* cycle with $2m+1$ edges admits a maximum matching with m edges and a minimum vertex cover with $m+1$ vertices. It seems that it's the odd-length cycles that results in a gap. In fact, this is true. Following we will show that, if an undirected graph G does not contain odd-length cycle, then the size of its maximum matching must be equal to the size of its minimum vertex cover. The proof is again constructive: we will design an algorithm to find both (the algorithm is an application of max-flow), and show that they have equal size.

We give an equivalent (and nice) characterization of an undirected graph without odd-length cycles.

An *bipartite* graph, usually written as $G=(X \cup Y, E)$, is an undirected graph whose vertices can be separated into two disjoint subsets X and Y such that all its edges must span X and Y (i.e., every edge $e=(u,v) \in E$ must satisfy $u \in X$ and $v \in Y$). See Figure 4.



$G=(X \cup Y, E)$ matching $M=\{(b,f),(c,g),(e,h)\}$ vertex cover $V_1=\{c,f,h\}$

Figure 4: A bipartite graph $G=(X \cup Y, E)$, a maximum matching M, and a minimum vertex cover V_1 of G.

Fact 1. An undirected graph G does not contain odd-length cycle if and only if G is a *bipartite* graph.

Proof. We first prove that if G is bipartite then any cycle in it must be even-length. (See cycle (b,h,c,f) in Figure 4.) This is because any cycle need to visit X and Y alternately so its length must be even.

For the other side, assume that every cycle of undirected graph G is even and we now prove that G is

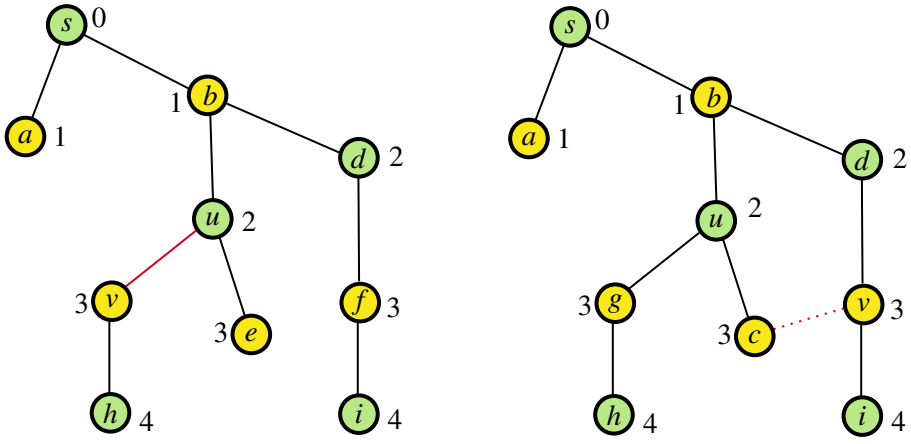


Figure 5: The shortest path tree T where vertices are assigned to X or Y based on the parity of the distance from s. Left: an edge $(u,v) \in E$ is part of the tree. Right: an edge $(u,v) \in E$ is not part of the tree.

bipartite. We assume that G consists of one connected component; otherwise all connected components of G can be processed separately in the same way below. Let s be any vertex in G. Let $distance(s,v)$ as the length of the shortest path from s to v. Let T be the shortest path tree wrt s, i.e., the number of edges on the path from s to v is exactly $distance(s,v)$. See Figure 5. We now put $v \in V$ into X if $distance(s,v)$ is even, and put $v \in C$ into Y if $distance(s,v)$ is odd. Clearly, X and Y is a partition of all vertices. We now show that, any edge $(u,v) \in E$ must span X and Y. There are two cases. The first one is that (u,v) is also an edge in T. In this case we must have that $distance(s,v) = distance(s,u) + 1$ so u and v must be assigned differently. (See Figure 5, Left.) The second case is that $(u,v) \notin T$. Suppose conversely that $distance(s,u)$ and $distance(s,v)$ have the same parity. Consider the s-u path and the s-v path in T. Let b be the branching point of these two paths (also called the least common ancestor of u and v). Then the b-u path and the b-v path are edge-disjoint and their lengths must also have the same parity. Now these two paths combined with edge (u,v) must form a cycle of odd-length in G, a contradiction. (See Figure 5, Right.) □