CS 341 Algorithms. Spring 2025. University of Waterloo

Lecture 15: Maximum flow and minimum cut

We first introduce the concepts of flows and cuts in a directed graph.

Then we study the Ford-Fulkerson algorithm for finding a maximum flow and the celebrated max-flow-min-cut theorem.

Flows and Paths

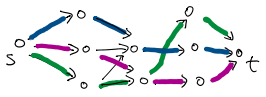
Using BFS / DFS, we can determine whether there is a path from vertex s to vertex t in linear time.

It is natural to consider the problem of finding 2 edge-disjoint paths from s to t, and more generally as many edge-disjoint paths from s to t as possible.

Edge-Disjoint Paths

Input: A directed graph $G=(V,E)$, and two vertices $s,t \in V$.

Output: A maximum number of edge-disjoint paths from s to t.



(Two paths P_1 and P_2 are edge-disjoint if they do not share an edge.)

This is basically the maximum flow problem, where the union of the paths defines a "flow" from s to t.

The maximum flow problem is defined on edge-capacitated graph, and the concept of flows is easier to work with than edge-disjoint paths.

s-t Flows

Let $G=(V,E)$ be a directed graph, where each edge $e \in E$ has an integer capacity $c(e) > 0$.

An s-t flow assigns a number $f(e)$ to each edge and satisfies the following two properties:

(i) Capacity constraints: $0 \le f(e) \le c(e)$ for each edge $e \in E$.

(ii) Flow conservation constraints: $f^{in}(v) = f^{out}(v)$ for each vertex $v \in V - s - t$ other than s and t, where

$f^{in}(v) = \sum_{e \text{ into } v} f(e)$ is the total incoming flow into v and $f^{out}(v) = \sum_{e \text{ out of } v} f(e)$ is the total outgoing flow from v.

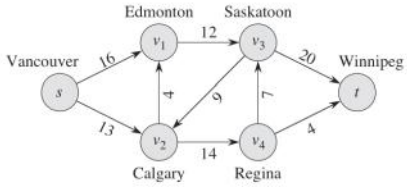
The value of the flow is defined as $\text{value}(f) = f^{out}(s) - f^{in}(s)$, the net flow going out of the source s.

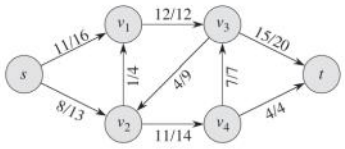
(In our algorithms, there won't be incoming flows to s, and so value(f) would simply be $f^{out}(s)$.)

Maximum s-t Flow

Input: A directed graph $G=(V,E)$ with a capacity $c(e)$ on each edge $e \in E$, and two vertices $s,t \in V$.

Output: A flow from s to t with maximum value.





[CLRS]

As you may imagine, this problem is fundamental and useful for modeling problems such as liquids flowing through pipes, currents through electrical networks, and information through communication networks.

But it turns out that it is also useful for solving seemingly unrelated problems that we'll see next time.

In the above example for edge-disjoint paths where every edge is of capacity one, the maximum flow from s to t is 3, by setting the flow on each colored edge to be one.

The connection between flows and paths is that a flow can always be decomposed into "capacity-disjoint" paths.

Lemma (decomposition of s-t flow into capacity-disjoint s-t paths)

Let f be an s-t flow where $f(e)$ is an integer for each edge e, with $f^{in}(s)=0$ and $value(f)=k$.

Then there are s-t paths $P_1, P_2, \ldots, P_k$ such that each edge e appears in $f(e)$ of these paths.

(For example, if $f(e)=0$, then the edge e will not appear in any of $P_1, \ldots, P_k$, while if $f(e)=2$, then e will appear in exactly 2 of these k paths.)

Proof (Sketch) The proof is by a simple induction. The base case is when $k=1$, where the statement holds.

Consider the edges with non-zero flow value. There must exist an s-t path P in these edges (why?).

Decrease the flow of each edge in P by one. i.e. $f'(e) := f(e)-1$ if $e \in P$, otherwise $f'(e) := f(e)$.

Check that f' is a flow with value $k-1$, as each vertex other that s,t still satisfies conservation constraint.

By induction, there are s-t paths $P_1, P_2, \ldots, P_{k-1}$ so that each edge e appears in $f'(e)$ of these $k-1$ paths.

Then, it should be clear that $P_1, P_2, \ldots, P_{k-1}, P$ are k paths that the lemma guarantees. □

For example, when the flow value is k and $f(e) \in \{0,1\}$ $\forall e \in E$, then the above lemma says that the flow f can be decomposed into k edge-disjoint paths.

So, even our goal is to find k edge-disjoint paths, it would be easier for us to focus on finding a flow of value k with $f(e) \in \{0,1\}$ $\forall e \in E$ instead, so that we don't need to worry about which edges belong to which paths during the algorithm, and only decompose the flow into edge-disjoint paths in the end.

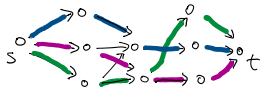
Exercise: Find a decomposition of the flow in the Canadian example above.

Flows and Cuts

What is a good upper bound on the value of a maximum flow? How do we know that a flow is maximum?

A trivial upper bound is the total capacity of all the edges, but it is almost never tight.

In the example for edge-disjoint paths, we know that there are at most 3 edge-disjoint s-t paths because the out-degree of s is 3.



We consider a generalization of this upper bound to a subset of vertices.

s-t Cut

Let $G=(V,E)$ be a directed graph, where each edge $e \in E$ has an integer capacity $c(e) > 0$.

A subset of vertices $\emptyset \neq S \subset V$ is an s-t cut if $s \in S$ and $t \notin S$, i.e. the partition $(S, V-S)$ separates s and t.

The capacity of an s-t cut S is defined as $c^{out}(S) := \sum_{e \in \delta^{out}(S)} c(e)$, the total capacity of the edges going out of S, where $\delta^{out}(S) := \{ uv \in E \mid u \in S, v \notin S \}$ is the set of directed edges going out of S.

Upper bounding max-s-t-flow by s-t cut

In the edge-disjoint paths problem where every edge is of capacity one, the capacity of an s-t cut S is simply the number of edges in $\delta^{out}(S)$. Note that the outdegree of s is the special case when $S = \{s\}$.

If an s-t cut S has at most k edges in $\delta^{out}(S)$, then it should be clear that there cannot be more than k edge-disjoint s-t paths, because each s-t path must have at least one edge in $\delta^{out}(S)$.

This idea can be generalized to maximum flow: If an s-t cut S has capacity k, then the value of a maximum s-t flow must be at most k.

One way to see this is through the decomposition of flow into paths in the lemma above. We leave it as an exercise.

We will give another proof of this upper bound later.

Minimum s-t Cut

To give the best upper bound on the value of a maximum s-t flow, we consider the minimum s-t cut problem.

Input: A directed graph $G=(V,E)$ with a capacity $c(e)$ on each edge $e \in E$, and two vertices $s, t \in V$.

Output: An s-t cut S with minimum capacity $c^{out}(S)$.

This is a natural and useful problem on its own, and we will see some interesting application next time.

Max-Flow Min-Cut Theorem

It turns out that having a small s-t cut is the only obstruction of having a large s-t flow, providing a concise and elegant proof of optimality for both problems.

Theorem The value of maximum s-t flow is equal to the capacity of a minimum s-t cut.

This is one of the most beautiful and important results in combinatorial optimization and graph theory.

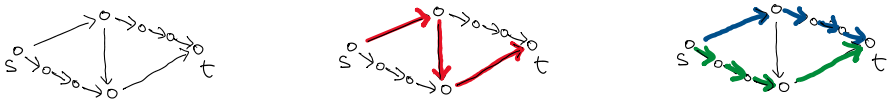
and has diverse applications in computer science and mathematics.

We will give an algorithmic proof of this theorem, that solves the maximum s-t flow problem and the minimum s-t cut problem at the same time.

Ford-Fulkerson Algorithm

A natural strategy to find edge-disjoint s-t paths is to say finding a shortest s-t path (so that it uses the minimum number of edges), then finding another shortest s-t path in the remaining graph, and so on.

But this approach does not necessarily obtain an optimal solution.

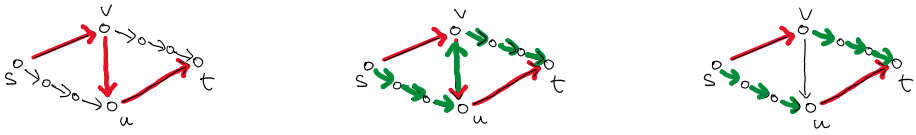


input graph | greedy solution | optimal solution

The difficulty of the greedy approach is to commit on a path that we have chosen, and there seems to be no good way to find an s-t path that belongs to an optimal solution.

The Ford-Fulkerson algorithm can be understood as a more general "local search" method, in which we may "undo" the decisions that we have made in earlier iterations, but only when we can improve the objective value of the solution (in this case the value of the flow).

For the above example, we will find a path and "push back" some of the flow to obtain the optimal solution.



greedy solution | "augmenting path" | optimal solution

In the middle picture, we send a flow from s to u, "undo / push-back" the flow from v to u in the greedy solution, and send a flow from v to t to obtain the optimal solution.

To define the augmenting path more precisely, it will be more convenient to introduce an auxiliary graph called the "residual graph", so that we don't need to distinguish between "push-forward / push-back".

Residual Graph

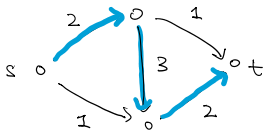
Given a directed graph G and a flow f on G, the residual graph G_f of G with respect to f is defined as follows:

- The vertex set of G_f is the same as the vertex set of G.
- For each edge e on which $f(e) < c(e)$, there are $c(e) - f(e)$ "leftover" units of capacity on which we can push flow forward. So we include the edge e in G_f, with a capacity $c(e) - f(e)$. We will call

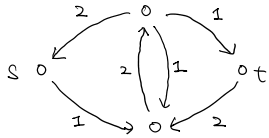
edges included this way the forward edges.

- For each edge $e = uv$ on which $f(e) > 0$, there are $f(e)$ units of flow that we can "undo" by pushing the flow backward. So we include the reverse edge $e' = vu$ in G_f, with a capacity of $f(e)$. We will call edges included this way the backward edges.

For an example,



a directed graph with a flow of 2 units



residual graph wrt the flow

Pushing Flow on an Augmenting Path

An augmenting path with respect to a flow f is just a simple s-t path in G_f.

Let P be an augmenting path wrt to f. Define bottleneck(P, f) as the minimum capacity of an edge on P in G_f.

The following is the subroutine to improve the flow f by using an augmenting path P.

```
augment (f, P)
    let b = bottleneck (P, f)
    for each edge e = uv on P
        if e is a forward edge
            increase f(e) by b in G
        else if e is a backward edge
            let e' = vu be the reverse edge
            decrease f(e') by b in G
```

We show that the subroutine augment(f, P) will always improve the value of the current flow f.

Lemma Let f be a flow in G with $f^{in}(s) = 0$, and P be an augmenting path with respect to f. Let f' be the resulting flow after the subroutine augment(f, P) is called. Then f' is a flow with value(f') = value(f) + bottleneck(P, f) and $f'^{in}(s) = 0$.

Proof First, we check that f' is a flow, for which we need to check the capacity and conservation constraints.

It should be clear by construction that f' still satisfies the capacity constraint in G.

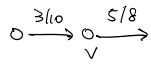
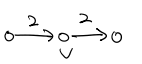
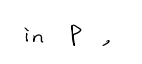
If e is a forward edge, then $0 \le f'(e) = f(e) + \text{bottleneck}(P, f) \le f(e) + (c(e) - f(e)) \le c(e)$;

while if e is a backward edge, then $c(e') \ge f'(e') = f(e') - \text{bottleneck}(P, f) \ge f(e') - f(e') \ge 0$,

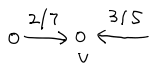
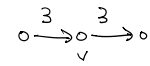
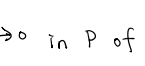
where e' is the reverse edge of e (i.e. if $e = uv$, then $e' = vu$).

For the conversation constraints, we verify that the change of the amount of flow entering v is equal to the change of the amount of flow leaving v, for any $v \in V - s - t$.

There are 4 cases to check, depending on whether the edges entering / exiting v are forward / backward edges.

Case 1: forward / forward  in f,  in P,  in f'

In this case, both the incoming flow to v and the outgoing flow from v are increased by b.

Case 2: forward / backward  in f,  in P of G_f, 

In this case, both the incoming flow to v and the outgoing flow from v are unchanged.

Case 3: backward / forward, Similar to case 2 with both unchanged.

Case 4: backward / backward, Similar to case 1 with both decreased by b.

Since conservation constraints were satisfied in f, they continue to be satisfied in f'.

Finally, we check that $\text{value}(f') = \text{value}(f) + \text{bottleneck}(P, f)$.

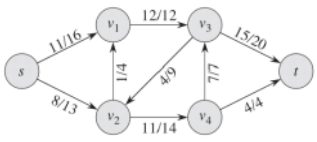
Since $f^{in}(s) = 0$, there are no backward edges incident on s in G_f.

Since P is a simple s-t path in G_f, the vertex s is only visited once with an outgoing forward edge.

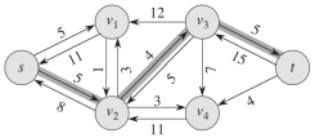
Therefore, by augmenting f with P, we maintain the property that $f'^{in}(s) = 0$ and thus

$$\text{value}(f') = f'^{out}(s) - f'^{in}(s) = f'^{out}(s) = f^{out}(s) + b = f^{out}(s) - f^{in}(s) + b = \text{value}(f) + b. \quad \square$$

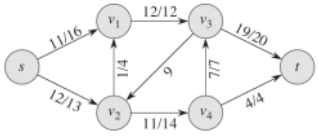
Example: [CLRS]



flow f



residual graph G_f and P



new flow $f' = \text{augment}(f, P)$

Remark: It is in the above lemma where it is easier to maintain a flow than to maintain a path decomposition, as for a flow we only need to check capacity and conservation constraints.

Ford-Fulkerson Algorithm

The above lemma shows that if we can find an augmenting path P in the residual graph G_f of the current flow f, then we can use it to obtain a flow f' with larger value.

The Ford-Fulkerson algorithm is simply to repeat this operation until we cannot do so.

```
Initially, f(e) = 0 for all edges e in G
while there is an s-t path P in G_f do
    f ← augment(f, P) and update the residual graph G_f.
```

This is it. In the next section, we will prove that this "local search" method always gives us

a maximum flow, i.e. whenever there is no augmenting path, the current flow is a maximum flow.

Max-Flow-Min-Cut Theorem

Our strategy to prove that Ford-Fulkerson algorithm always finds a maximum flow is to show that when there is no augmenting path, there is a cut with capacity equal to the value of the current flow, thereby proving the max-flow min-cut theorem as well as solving the minimum s-t cut problem simultaneously.

To do so, we need the following claim about the flow of any s-t cut, from which the easy direction of the max-flow min-cut theorem (i.e. max flow $\leq$ min cut) follows immediately.

Claim Let f be any s-t flow, and $\phi \neq S \subset V$ be any s-t cut (i.e. $s \in S$ and $t \notin S$).

Then $f^{out}(S) - f^{in}(S) = value(f) = f^{out}(s) - f^{in}(s)$.

Proof The intuition is that only the source vertex s has a positive "net output", while all other vertices in $V-s-t$ satisfy the flow conservation constraints (intuitively just passing the flow along), and so the "net output" of an s-t cut S is simply equal to the "net output" of s.

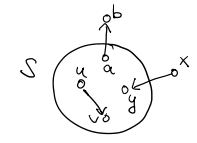
The proof is by a subtle manipulation:

$$value(f) = f^{out}(s) - f^{in}(s) = \sum_{v \in S} \left(f^{out}(v) - f^{in}(v) \right) \quad // \text{ because } f^{out}(v) - f^{in}(v) = 0 \ \forall v \in V-s-t$$

$$= \sum_{v \in S} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \right) \quad // \text{ by definition}$$

$$= \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e)$$

$$= f^{out}(S) - f^{in}(S).$$



To see the second last equality, divide the edges involving in the sum into three types.

Note that the edges involving in the sum in the second line must have at least one vertex in S.

(1) first type: $e = uv$, $u \in S$, $v \in S$. For this edge e, $f(e)$ appears positively in $f^{out}(u)$, and $f(e)$ appears negatively in $f^{in}(v)$, and so this edge contributes zero to the sum in the second line.

(2) second type: $e = ab$, $a \in S$, $b \notin S$, so ab is an outgoing edge of S.

For this edge e, it only appears once positively in $f^{out}(a)$, and so contributes $f(e)$ to the sum.

(3) third type: $e = xy$, $x \notin S$, $y \in S$, so xy is an incoming edge of S.

For this edge e, it only appears once negatively in $f^{in}(y)$, and so contributes $-f(e)$ to the sum.

This verifies the second last equality and thus completes the proof. □

Corollary Let f be any s-t flow, and $\phi \neq S \subset V$ be any s-t cut (i.e. $s \in S$ and $t \notin S$).

Then $value(f) = f^{out}(S) - f^{in}(S) \leq f^{out}(S) \leq c^{out}(S)$.

This is the easy direction of the max-flow min-cut theorem: the value of any s-t flow is at most

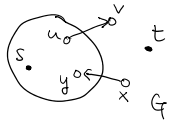
the capacity of any s-t cut, thus the value of a max s-t flow is at most the capacity of a min s-t cut.

We are ready to prove the correctness of the Ford-Fulkerson algorithm and the max-flow-min-cut theorem.

Proposition. If f is an s-t flow such that there is no s-t path in the residual graph G_f, then there is an s-t cut S such that $\text{value}(f) = c^{out}(S)$.

Consequently, f has the maximum value of any s-t flow, and S has the minimum capacity of any s-t cut.

Proof As there is no path from s to t in G_f, by the result in BFS/DFS, there exists a subset S with $s \in S$, $t \notin S$ such that S has no outgoing edges in G_f. (Recall this?)



We claim that $c^{out}(S) = \text{value}(f)$. Consider two types of edges.

(1) Consider an edge $uv \in \delta^{out}(S)$ in G, with $u \in S$ and $v \notin S$.

Since S has no outgoing edge in G_f, we must have $uv \notin G_f$.

This implies that $f(uv) = c(uv)$, as otherwise uv would be a forward edge in G_f.

(2) Consider an edge $xy \in \delta^{in}(S)$ in G, with $x \notin S$ and $y \in S$.

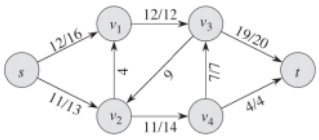
Since S has no outgoing edge in G_f, we must have $yx \notin G_f$.

This implies that $f(xy) = 0$, as otherwise yx would be a backward edge in G_f.

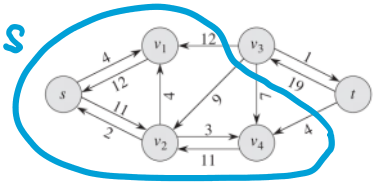
Putting together, it follows that $f^{out}(S) - f^{in}(S) = c^{out}(S) - 0 = c^{out}(S)$.

By the claim above, we conclude that $\text{value}(f) = f^{out}(S) - f^{in}(S) = c^{out}(S)$. □

Example: [CLRS]



maximum s-t flow f in G



residual graph G_f and minimum s-t cut

Note that this provides the correctness proof of the Ford-Fulkerson algorithm, as well as providing an algorithmic proof of the max-flow min-cut theorem.

Furthermore, this proof provides a linear time algorithm to compute a minimum s-t cut from a maximum s-t flow.

We will analyze the time complexity of the Ford-Fulkerson algorithm in the next section.

Time Complexity

It is not difficult to analyze the time complexity of the Ford-Fulkerson algorithm when all capacities are integers.

Claim Let $G = (V, E)$ be a directed graph where $c(e)$ is an integer for every $e \in E$.

Let k be the value of a maximum s-t flow. The Ford-Fulkerson algorithm terminates in $O(k \cdot |E|)$ time.

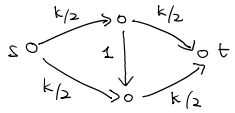
Proof Each iteration can be implemented in $O(|E|)$ time, as searching an s-t path in G_f can be done by BFS/DFS.

If all capacities are integers, then bottleneck(P,f) is at least one.

This implies that the flow value is increased by at least one after each augmentation, and so there are at most k iterations in the algorithm. □

For the edge-disjoint s-t path problem in a simple directed graph, $k \le |V|$ and so the Ford-Fulkerson algorithm is a $O(|V|\cdot|E|)$-time algorithm for the problem.

The analysis of the above claim is tight, as there are examples where the algorithm takes k iterations.

For example, in 

, there could be k iterations of augmentation (do you see why?).

So, in general, when k is large, the Ford-Fulkerson algorithm could be very slow.

Real-valued capacities: You may wonder why we made the assumption that all edge capacities are integers.

Surprisingly, if we allow the edge capacities be real numbers, then there are pathological examples where the Ford-Fulkerson algorithm can run forever with pathological choices of augmenting paths.

Choosing good augmenting paths: So a natural question is whether we can choose augmenting paths in a good way that avoids these pathological examples.

Interestingly, perhaps the most natural implementation of the Ford-Fulkerson algorithm would not have any issue.

Edmonds and Karp proved that if we always find a shortest s-t path in the residual graph G_f, then the resulting algorithm (which we call Edmonds-Karp algorithm) terminates in $O(|V||E|)$ iterations, even when the edge capacities are real-valued.

So, if we just implement the Ford-Fulkerson algorithm using BFS (which finds shortest s-t path), then the time complexity is at most $O(|V|^2\cdot|E|)$.

Another natural strategy is to choose an augmenting path P that maximizes bottleneck(P,f).

A modification of Dijkstra's algorithm can find such a path in $O(|E|\log|V|)$ time (exercise!).

If all edge capacities are integers, then it is not difficult to argue that there are at most $O(|E|\cdot\log k)$ iterations where k is the value of a maximum flow.

We won't go into details. You should be able to learn all these in the "Network Flows" course in C&O.

Concluding Remarks

There is a vast literature in designing faster algorithms for the maximum flow problem.

Very recently, in 2022, researchers finally found an almost linear-time algorithm for maximum flow (see the paper "Maximum flow and minimum cost flow in almost linear time").

Traditionally, maximum flow is solved using combinatorial techniques and sophisticated data structures.

In the past decade or so, however, researchers used a lot of tools and ideas from convex optimization to design faster algorithms for combinatorial problems.

The recent breakthrough is based on both convex optimization and sophisticated modern data structures.

As a related note, the maximum flow problem can be formulated as a linear programming problem.

Linear programming is a general framework to solve many combinatorial optimization problems.

The Ford-Fulkerson algorithm can be understood as a special instantiation of the simplex algorithm for solving linear programming problems.

The local search method is a general phenomenon for linear programming problems and more generally convex optimization problems, in which local minimums are global minimums.

You can take a look at chapter 7 of [DPV] for an introduction of this general framework, which is one of the most important paradigms in algorithm design.

References: [KT 7.1-7.3], [CLRS 26.1-26.2]