

CS261: A Second Course in Algorithms

Lecture #1: Course Goals and Introduction to Maximum Flow*

Tim Roughgarden[†]

January 5, 2016

1 Course Goals

CS261 has two major course goals, and the course splits roughly in half along these lines.

1.1 Well-Solved Problems

This first goal is very much in the spirit of an introductory course on algorithms. Indeed, the first few weeks of CS261 are pretty much a direct continuation of CS161 — the topics that we'd cover at the end of CS161 at a semester school.

Course Goal 1 Learn efficient algorithms for fundamental and well-solved problems.

There's a collection of problems that are flexible enough to model many applications and can also be solved quickly and exactly, in both theory and practice. For example, in CS161 you studied shortest-path algorithms. You should have learned all of the following:

1. The formal definition of one or more variants of the shortest-path problem.
2. Some famous shortest-path algorithms, like Dijkstra's algorithm and the Bellman-Ford algorithm, which belong in the canon of algorithms' greatest hits.
3. Applications of shortest-path algorithms, including to problems that don't explicitly involve paths in a network. For example, to the problem of planning a sequence of decisions over time.

*©2016, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

The study of such problems is top priority in a course like CS161 or CS261. One of the biggest benefits of these courses is that they prevent you from reinventing the wheel (or trying to invent something that doesn't exist), instead allowing you to stand on the shoulders of the many brilliant computer scientists who preceded us. When you encounter such problems, you already have good algorithms in your toolbox and don't have to design one from scratch. This course will also give you practice spotting applications that are just thinly disguised versions of these problems.

Specifically, in the first half of the course we'll study:

1. the maximum flow problem;
2. the minimum cut problem;
3. graph matching problems;
4. linear programming, one the most general polynomial-time solvable problems known.

Our algorithms for these problems will have running times a bit bigger than those you studied in CS161 (where almost everything runs in near-linear time). Still, these algorithms are sufficiently fast that you should be happy if a problem that you care about reduces to one of these problems.

1.2 Not-So-Well-Solved Problems

Course Goal 2 Learns tools for tackling not-so-well-solved problems.

Unfortunately, many real-world problems fall into this camp, for many different reasons. We'll focus on two classes of such problems.

1. *NP*-hard problems, for which we don't expect there to be any exact polynomial-time algorithms. We'll study several broadly useful techniques for designing and analyzing heuristics for such problems.
2. Online problems. The anachronistic name does not refer to the Internet or social networks, but rather to the realistic case where an algorithm must make irrevocable decisions without knowing the future (i.e., without knowing the whole input).

We'll focus on algorithms for *NP*-hard and online problems that are guaranteed to output a solution reasonably close to an optimal one.

1.3 Intended Audience

CS261 has two audiences, both important. The first is students who are taking their final algorithms course. For this group, the goal is to pack the course with essential and likely-to-be-useful material. The second is students who are contemplating a deeper study of algorithms. With this group in mind, when the opportunity presents itself, we'll discuss

recent research developments and give you a glimpse of what you'll see in future algorithms courses. For this second audience, CS261 has a third goal.

Course Goal 3 Provide a gateway to the study of advanced algorithms.

After completing CS261, you'll be well equipped to take any of the many 200- and 300-level algorithms courses that the department offers. The pace and difficulty level of CS261 interpolates between that of CS161 and more advanced theory courses.

When you speak to audience, it's good to have one or a few "canonical audience members" in mind. For your reference and amusement, here's your instructor's mental model for canonical students in courses at different levels:

1. CS161: a constant fraction of the students do not want to be there, and/or hate math.
2. CS261: a self-selecting group of students who like algorithms and want to learn much more about them. Students may or may not love math, but they shouldn't hate it.
3. CS3xx: geared toward students who are doing or would like to do research in algorithms.

2 Introduction to the Maximum Flow Problem

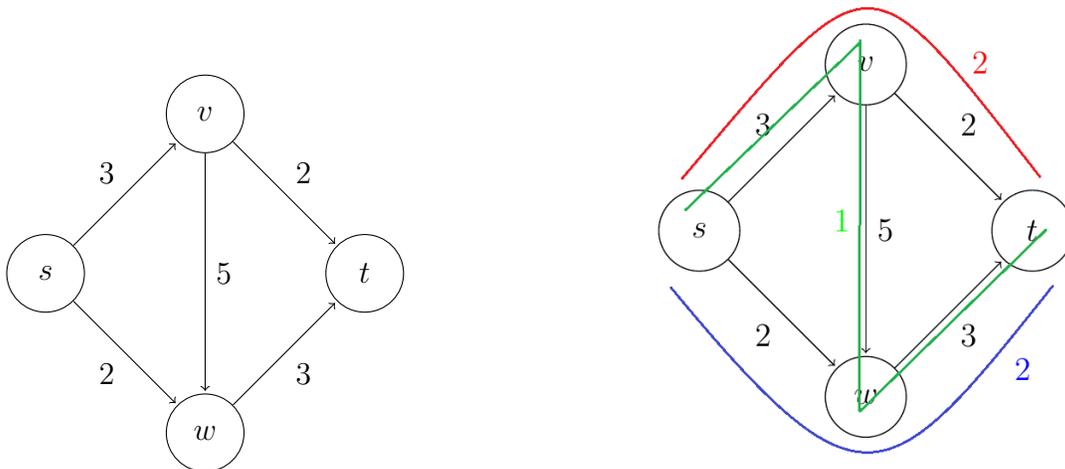


Figure 1: (a, left) Our first flow network. Each edge is associated with a capacity. (b, right) A sample flow of value 5, where the red, green and blue paths have flow values of 2, 1, 2 respectively.

2.1 Problem Definition

The maximum flow problem is a stone-cold classic in the design and analysis of algorithms. It's easy to understand intuitively, so let's do an informal example before giving the formal

definition.

The picture in Figure 1(a) resembles the ones you saw when studying shortest paths, but the semantics are different. Each edge is labeled with a *capacity*, the maximum amount of stuff that it can carry. The goal is to figure out how much stuff can be pushed from the vertex s to the vertex t .

For example, Figure 1(b) exhibits a method of pushing five units of flow from s to t , while respecting all edges' capacities. Can we do better? Certainly not, since at most 5 units of flow can escape s on its two outgoing edges.

Formally, an instance of the maximum flow problem is specified by the following ingredients:

- a directed graph G , with vertices V and directed edges E ;¹
- a *source* vertex $s \in V$;
- a *sink* vertex $t \in V$;
- a nonnegative and integral capacity u_e for each edge $e \in E$.

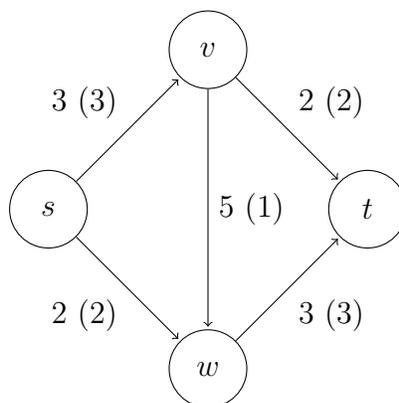


Figure 2: Denoting a flow by keeping track of the amount of flow on each edge. Flow amount is given in brackets.

Since the point is to push flow from s to t , we can assume without loss of generality that s has no incoming edges and t has no outgoing edges.

Given such an input, the feasible solutions are the *flows* in the network. While Figure 1(b) depicts a flow in terms of several paths, for algorithms, it works better to just keep track of the amount of flow on each edge (as in Figure 2).² Formally, a flow is a nonnegative vector $\{f_e\}_{e \in E}$, indexed by the edges of G , that satisfies two constraints:

¹All of our maximum flow algorithms can be extended to undirected graphs; see Exercise Set #1.

²Every flow in this sense arises as the superposition of flow paths and flow cycles; see Problem #1.

Capacity constraints: $f_e \leq u_e$ for every edge $e \in E$;

Conservation constraints: for every vertex v other than s and t ,

$$\text{amount of flow entering } v = \text{amount of flow exiting } v.$$

The left-hand side is the sum of the f_e 's over the edge incoming to v ; likewise with the outgoing edges for the right-hand side.

The objective is to compute a *maximum flow*— a flow with the maximum-possible *value*, meaning the total amount of flow that leaves s . (As we'll see, this is the same as the total amount of flow that enters t .)

2.2 Applications

Why should we care about the maximum flow problem? Like all central algorithmic problems, the maximum flow problem is useful in its own right, plus many different problems are really just thinly disguised version of maximum flow. For some relatively obvious and literal applications, the maximum flow problem can model the routing of traffic through a transportation network, packets through a data network, or oil through a distribution network.³ In upcoming lectures we'll prove the less obvious fact that problems ranging from bipartite matching to image segmentation reduce to the maximum flow problem.

2.3 A Naive Greedy Algorithm

We now turn our attention to the design of efficient algorithms for the maximum flow problem. A priori, it is not clear that any such algorithms exist (for all we know right now, the problem is *NP*-hard).

We begin by considering greedy algorithms. Recall that a greedy algorithm is one that makes a sequence of myopic and irrevocable decisions, with the hope that everything somehow works out at the end. For most problems, greedy algorithms do not generally produce the best-possible solution. But it's still worth trying them, because the ways in which greedy algorithms break often yields insights that lead to better algorithms.

The simplest greedy approach to the maximum flow problem is to start with the all-zero flow and greedily produce flows with ever-higher value. The natural way to proceed from one to the next is to send more flow on some path from s to t (cf., Figure 1(b)).

³A flow corresponds to a steady-state solution, with a constant rate of arrivals at s and departures at t . The model does not capture the time at which flow reaches different vertices. However, it's not hard to extend the model to also capture temporal aspects as well.

A Naive Greedy Algorithm

```

initialize  $f_e = 0$  for all  $e \in E$ 
repeat
  search for an  $s$ - $t$  path  $P$  such that  $f_e < u_e$  for every  $e \in P$ 
  // takes  $O(|E|)$  time using BFS or DFS
  if no such path then
    halt with current flow  $\{f_e\}_{e \in E}$ 
  else
    let  $\Delta = \min_{e \in P} \underbrace{(u_e - f_e)}_{\text{room on } e}$ 
    for all edges  $e$  of  $P$  do
      increase  $f_e$  by  $\Delta$ 
  
```

Note that the path search just needs to determine whether or not there is an s - t path in the subgraph of edges e with $f_e < u_e$. This is easily done in linear time using your favorite graph search subroutine, such as breadth-first or depth-first search. There may be many such paths; for now, we allow the algorithm to choose one arbitrarily. The algorithm then pushes as much flow as possible on this path, subject to capacity constraints.

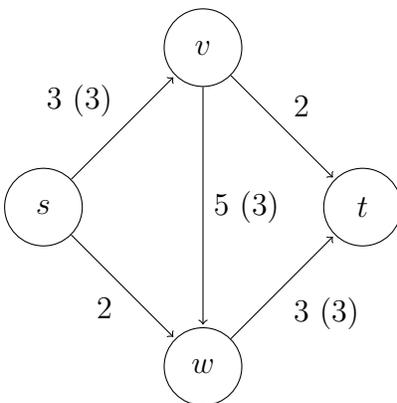


Figure 3: Greedy algorithm returns suboptimal result if first path picked is s - v - w - t .

This greedy algorithm is natural enough, but does it work? That is, when it terminates with a flow, need this flow be a maximum flow? Our sole example thus far already provides a negative answer (Figure 3). Initially, with the all-zero flow, all s - t paths are fair game. If the algorithm happens to pick the zig-zag path, then $\Delta = \min\{3, 5, 3\} = 3$ and it routes 3 units of flow along the path. This saturates the upper-left and lower-right edges, at which point there is no s - t path such that $f_e < u_e$ on every edge. The algorithm terminates at this

point with a flow with value 3. We already know that the maximum flow value is 5, and we conclude that the naive greedy algorithm can terminate with a non-maximum flow.⁴

2.4 Residual Graphs

The second idea is to extend the naive greedy algorithm by allowing “undo” operations. For example, from the point where this algorithm gets stuck in Figure 3, we’d like to route two more units of flow along the edge (s, w) , then *backward* along the edge (v, w) , undoing 2 of the 3 units we routed the previous iteration, and finally along the edge (v, t) . This would yield the maximum flow of Figure 1(b).



Figure 4: (a) original edge capacity and flow and (b) resultant edges in residual network.

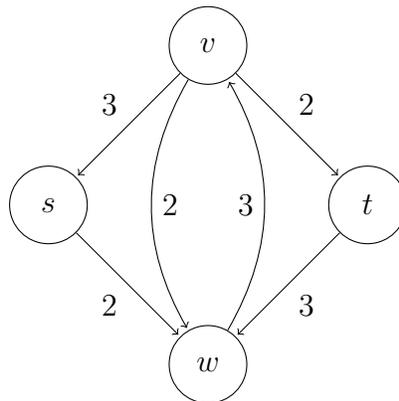


Figure 5: Residual network of flow in Figure 3.

We need a way of formally specifying the allowable “undo” operations. This motivates the following simple but important definition, of a *residual network*. The idea is that, given a graph G and a flow f in it, we form a new flow network G_f that has the same vertex set of G and that has two edges for each edge of G . An edge $e = (v, w)$ of G that carries flow f_e and has capacity u_e (Figure 4(a)) spawns a “forward edge” (v, w) of G_f with capacity $u_e - f_e$ (the room remaining) and a “backward edge” (w, v) of G_f with capacity f_e (the amount

⁴It does compute what’s known as a “blocking flow;” more on this next lecture.

of previously routed flow that can be undone). See Figure 4(b).⁵ Observe that s - t paths with $f_e < u_e$ for all edges, as searched for by the naive greedy algorithm, correspond to the special case of s - t paths of G_f that comprise only forward edges.

For example, with G our running example and f the flow in Figure 3, the corresponding residual network G_f is shown in Figure 5. The four edges with zero capacity in G_f are omitted from the picture.⁶

2.5 The Ford-Fulkerson Algorithm

Happily, if we just run the natural greedy algorithm in the current residual network, we get a correct algorithm, the *Ford-Fulkerson algorithm*.⁷

Ford-Fulkerson Algorithm

```

initialize  $f_e = 0$  for all  $e \in E$ 
repeat
  search for an  $s$ - $t$  path  $P$  in the current residual graph  $G_f$  such that
    every edge of  $P$  has positive residual capacity
  // takes  $O(|E|)$  time using BFS or DFS
  if no such path then
    halt with current flow  $\{f_e\}_{e \in E}$ 
  else
    let  $\Delta = \min_{e \in P}$  ( $e$ 's residual capacity in  $G_f$ )
    // augment the flow  $f$  using the path  $P$ 
    for all edges  $e$  of  $G$  whose corresponding forward edge is in  $P$  do
      increase  $f_e$  by  $\Delta$ 
    for all edges  $e$  of  $G$  whose corresponding reverse edge is in  $P$  do
      decrease  $f_e$  by  $\Delta$ 

```

For example, starting from the residual network of Figure 5, the Ford-Fulkerson algorithm will augment the flow by units along the path $s \rightarrow w \rightarrow v \rightarrow t$. This augmentation produces the maximum flow of Figure 1(b).

We now turn our attention to the correctness of the Ford-Fulkerson algorithm. We'll worry about optimizing the running time in future lectures.

⁵If G already has two edges (v, w) and (w, v) that go in opposite directions between the same two vertices, then G_f will have two parallel edges going in either direction. This is not a problem for any of the algorithms that we discuss.

⁶More generally, when we speak about "the residual graph," we usually mean after all edges with zero residual capacity have been removed.

⁷Yes, it's the same Ford from the Bellman-Ford algorithm.

2.6 Termination

We claim that the Ford-Fulkerson algorithm eventually terminates with a feasible flow. This follows from two invariants, both proved by induction on the number of iterations.

First, the algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow. This is clearly true initially. The parameter Δ is defined so that no flow value f_e becomes negative or exceeds the capacity u_e . For the conservation constraints, consider a vertex v . If v is not on the augmenting path P in G_f , then the flow into and out of v remain the same. If v is on P , with edges (x, v) and (v, w) belonging to P , then there are four cases, depending on whether or not (x, v) and (v, w) correspond to forward or reverse edges. For example, if both are forward edges, then the flow augmentation increases both the flow into and the flow out of v increase by Δ . If both are reverse edges, then both the flow into and the flow out of v decrease by Δ . In all four cases, the flow in and flow out change by the same amount, so conservation constraints are preserved.

Second, the Ford-Fulkerson algorithm maintains the property that every flow amount f_e is an integer. (Recall we are assuming that every edge capacity u_e is an integer.) Inductively, all residual capacities are integral, so the parameter Δ is integral, so the flow stays integral.

Every iteration of the Ford-Fulkerson algorithm increase the value of the current flow by the current value of Δ . The second invariant implies that $\Delta \geq 1$ in every iteration of the Ford-Fulkerson algorithm. Since only a finite amount of flow can escape the source vertex, the Ford-Fulkerson algorithm eventually halts. By the first invariant, it halts with a feasible flow.⁸

Of course, all of this applies equally well to the naive greedy algorithm of Section 2.3. How do we know whether or not the Ford-Fulkerson algorithm can also terminate with a non-maximum flow? The hope is that because the Ford-Fulkerson algorithm has more path eligible for augmentation, it progresses further before halting. But is it guaranteed to compute a maximum flow?

2.7 Optimality Conditions

Answering the following question will be a major theme of the first half of CS261, culminating with our study of linear programming duality.

HOW DO WE KNOW WHEN WE'RE DONE?

For example, given a flow, how do we know if it's a maximum flow? Any correct maximum flow algorithm must answer this question, explicitly or implicitly. If I handed you an allegedly maximum flow, how could I convince you that I'm not lying? It's easy to convince someone that a flow is *not* maximum, just by exhibiting a flow with higher value.

⁸The Ford-Fulkerson algorithm continues to terminate if edges' capacities are rational numbers, not necessarily integers. (Proof: scaling all capacities by a common number doesn't change the problem, so we can clear denominators to reduce the rational capacity case to the integral capacity case.) It is a bizarre mathematical curiosity that the Ford-Fulkerson algorithm need not terminate with edges' capacities are irrational.

Returning to our original example (Figure 1), answering this question didn't seem like a big deal. We exhibited a flow of value 5, and because the total capacity escaping s is only 5, it's clear that there can't be any flow with high value. But what about the network in Figure 6(a)? The flow shown in Figure 6(b) has value only 3. Could it really be a maximum flow?

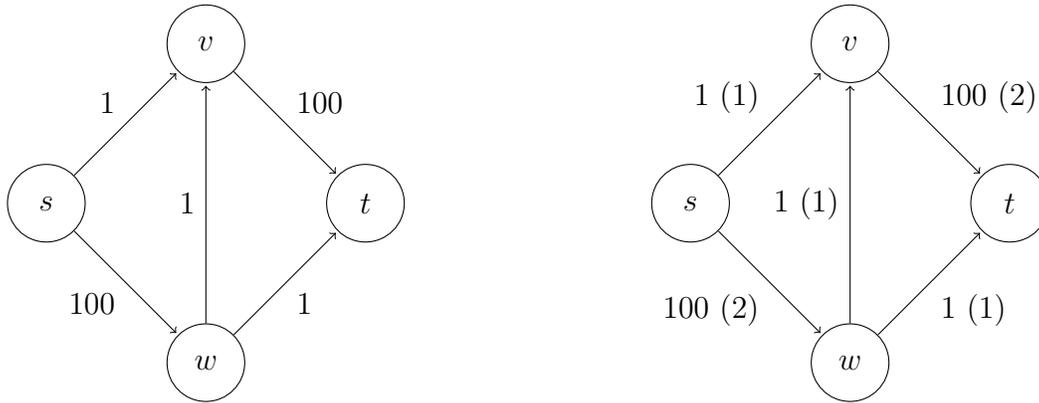


Figure 6: (a) A given network and (b) the alleged maximum flow of value 3.

We'll tackle several fundamental computational problems by following a two-step paradigm.

Two-Step Paradigm

1. Identify “optimality conditions” for the problem. These are sufficient conditions for a feasible solution to be an optimal solution. This step is structural, and not necessarily algorithmic. The optimality conditions vary with the problem, but they are often quite intuitive.
2. Design an algorithm that terminates with the optimality conditions satisfied. Such an algorithm is necessarily correct.

This paradigm is a guide for proving algorithms correct. Correctness proofs didn't get too much airtime in CS161, because almost all of them are straightforward inductions — think of MergeSort, or Dijkstra's algorithm, or any dynamic programming algorithm. The harder problems studied in CS261 demand a more sophisticated and principle approach (with which you'll get plenty of practice).

So how would we apply this two-step paradigm to the maximum flow problem? Consider the following claim.

Claim 2.1 (Optimality Conditions for Maximum Flow) *If f is a flow in G such that the residual network G_f has no s - t path, then the f is a maximum flow.*

This claim implements the first step of the paradigm. The Ford-Fulkerson algorithm, which can only terminate with this optimality condition satisfied, already provides a solution to the second step. We conclude:

Corollary 2.2 *The Ford-Fulkerson algorithm is guaranteed to terminate with a maximum flow.*

Next lecture we'll prove (a generalization of) the claim, derive the famous maximum-flow/minimum-cut problem, and design faster maximum flow algorithms.