

Lecture Note 8

Max Flow, Reductions, and Linear Programming

draft of September 29, 2026, by [N. Young](#)

© 2018–2026 [Neal E. Young](#). Text: CC BY-NC-SA 4.0; code: MIT; except where noted. See <https://github.com/nealeyoung/algs101/blob/main/LICENSE>.

Contents

8.1	Maximum Flow	1
8.1.1	The Minimum s - t Cut problem	2
8.1.2	A greedy Max-Flow algorithm that doesn't work	2
8.1.3	The Ford-Fulkerson algorithm	3
8.1.4	Consequences: The Max-Flow/Min-Cut Theorem; Integer Flows	5
8.1.5	Proof of Lemma 8.1	6
8.1.6	External sources on Max Flow	6
8.2	Polynomial-time reductions	7
8.2.1	Example 1: Bipartite Matching $\leq_P$ Max Flow	8
8.2.2	Example 2: Disjoint Paths $\leq_P$ Max Flow	10
8.2.3	Exercise: Bipartite Matching $\leq_P$ Disjoint Paths	11
8.2.4	The two key properties of polynomial-time reductions	12
8.2.5	External sources on polynomial-time reductions	13
8.3	Linear Programming and Integer Linear Programming	14
8.3.1	Max Flow as a linear program	15
8.3.2	Min Cut as a linear program	16
8.3.3	Shortest s - t path as a linear program	17
8.3.4	Subset Sum as an integer linear program	18
8.3.5	External sources on linear programming	18
8.4	Python code	20

8.1 Maximum Flow

The problem is defined as follows.

input: a *flow network* $(G = (V, E), \text{cap}, s, t)$, consisting of

digraph $G = (V, E)$; edge capacities $\text{cap} : E \rightarrow \mathbb{R}_+$; *source* $s \in V$ and *sink* $t \in V$

output: a *maximum s - t flow* f in G (defined below)

Fix any input $(G = (V, E), \text{cap}, s, t)$. What is an s - t flow? Imagine that there is a water source at s . Water must be sent from s , through the edges and vertices of the graph, to sink t . Each edge $(u, w) \in E$ can carry at most $\text{cap}(u, w)$ units of water per second from u to w . For each vertex v other than s or t , all water entering v must leave v . The goal is to maximize the rate at which water is sent.

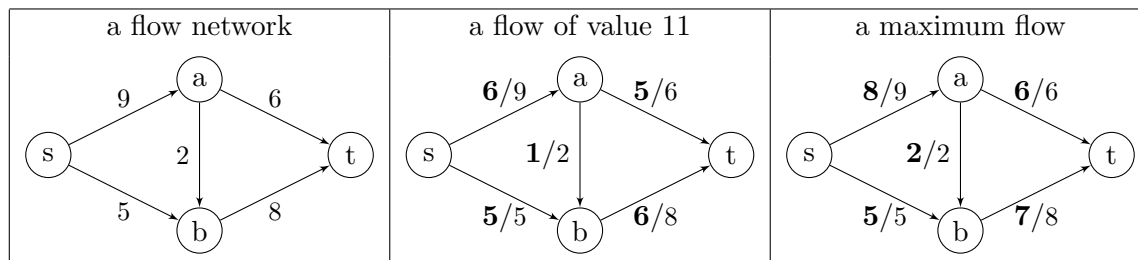
Formally, an s - t flow f in G is a function $f : E \rightarrow \mathbb{R}_+$, assigning a flow value $f(u, w) \geq 0$ to each edge $(u, w) \in E$, subject to the following constraints. For each edge $(u, w) \in E$, the flow $f(u, w)$ on the edge must be at most the capacity $\text{cap}(u, w)$. (These are called the *capacity* constraints.) For each vertex $v \in V \setminus \{s, t\}$ other than the source or sink, the flow on the edges into v must be equal to the flow on the edges leaving v . (These are called the *conservation* constraints.)

Any function $f : E \rightarrow \mathbb{R}_+$ meeting these constraints is called a *flow* (for the given flow network). The *value* (or *size*) of the flow, denoted $|f|$, is the net flow out of the source. (As we shall see, this necessarily equals the net flow into the sink.) The goal is to find a flow f of maximum value.

Here is a restatement of the problem. Let $\text{in}(v) = \{u \in V : (u, v) \in E\}$ denote the in-neighbors of v . Let $\text{out}(v) = \{w \in V : (v, w) \in E\}$ denote the out-neighbors. Using this notation, the problem is as follows:

$$\begin{aligned} \text{maximize } |f| &= \sum_{w \in \text{out}(s)} f(s, w) - \sum_{u \in \text{in}(s)} f(u, s) \quad \text{subject to:} \\ (\forall (u, w) \in E) \quad 0 &\leq f(u, w) \leq \text{cap}(u, w) \quad \text{--- capacity constraints} \\ (\forall v \in V \setminus \{s, t\}) \quad \sum_{w \in \text{out}(v)} f(v, w) &= \sum_{u \in \text{in}(v)} f(u, v) \quad \text{--- conservation constraints} \end{aligned}$$

Here is one example of a flow network, a flow in it, and a maximum flow in it:



8.1.1 The Minimum s - t Cut problem

input: a flow network $(G = (V, E), \text{cap}, s, t)$

output: an s - t cut of minimum *capacity* (terms defined below)

An *cut* is a partition of the vertex set V into two parts¹ (S, T) . Given any such cut, define $E(S, T)$ to be the set of edges crossing the cut from S to T . That is, $E(S, T) = \{(u, w) \in E : u \in S, w \in T\} = E \cap (S \times T)$. Define the *capacity* of the cut, $\text{cap}(S, T)$, to be the total capacity of those edges. For any flow f , define the *flow from S to T* , $f(S, T)$, to be the total flow on those edges. That is

$$\text{cap}(S, T) = \sum_{(u,w) \in E(S,T)} \text{cap}(u, w), \quad \text{and} \quad f(S, T) = \sum_{(u,w) \in E(S,T)} f(u, w).$$

In the previous example,

$$\text{cap}(\{s\}, \{a, b, t\}) = 9 + 5 = 14,$$

$$\text{cap}(\{s, b\}, \{a, t\}) = 17, \text{ and}$$

$$\text{cap}(\{s, a, b\}, \{t\}) = 6 + 8 = 14.$$

An s - t cut is a cut (S, T) with the additional property that $s \in S$ and $t \in T$. Next we'll show that the capacity of *any* s - t cut is an upper bound on the value of *any* s - t flow. To prove it we use the following useful property of flows:

Lemma 8.1. *Let f and (S, T) be any flow and s - t cut. Then $|f| = f(S, T) - f(T, S)$.*

That is, the value of the flow equals the flow from S to T minus the flow from T to S .

In the previous example, for the cut $S = \{s, b\}$, the flow f in the third panel has $f(\{s, b\}, \{a, t\}) - f(\{a, t\}, \{s, b\}) = (8 + 7) - 2 = 13 = |f|$.

Summing the conservation constraints over all vertices $v \in S \setminus \{s\}$ gives the equation in the lemma. You can find the detailed proof at the end of this note. The bound we mentioned follows:

Lemma 8.2. *Let f and (S, T) be any flow and s - t cut. Then $|f| \leq \text{cap}(S, T)$.*

That is, the value of any s - t flow is at most the capacity of any s - t cut.

Proof. Lemma 8.1 implies $|f| = f(S, T) - f(T, S) \leq f(S, T) \leq \text{cap}(S, T)$.

(The two inequalities hold because $f(T, S) \geq 0$, and f satisfies the capacity constraints.) □

In the previous example, the s - t cut $S = \{s, a\}$ has capacity 13. So, by the lemma, every s - t flow has value at most 13. The flow in the third panel achieves value 13, so must be a maximum flow.

8.1.2 A greedy Max-Flow algorithm that doesn't work

Consider the following greedy algorithm for Maximum Flow:

```

greedy-flow( $G = (V, E), \text{cap}, s, t$ )
1. initialize  $f[u, w] = 0$  for all  $(u, w) \in E$ 
2. for any path  $p$  in  $G$ , define  $\delta(p) = \min\{\text{cap}(u, w) - f[u, w] : (u, w) \in p\}$  ( $p$ 's residual capacity)
3. while there exists an  $s$ - $t$  path  $p$  with positive residual capacity  $\delta(p) > 0$ :
4.1. let  $p$  be any such path
4.2. for each edge  $(u, w) \in p$ : increase  $f[u, w]$  by  $\delta(p)$  (augment the flow along path  $p$ )
5. return  $f$ 
    
```

¹So $S \subseteq V$ and $T = V \setminus S$.

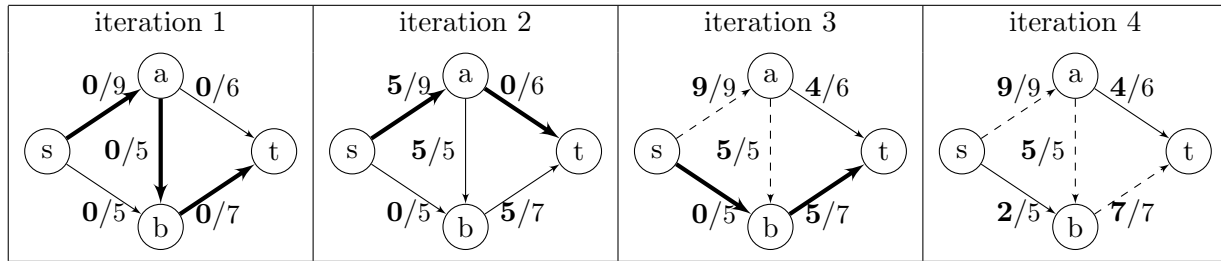


Figure 8.1: An example in which the greedy algorithm gets stuck.

In Step 3, the path p with positive residual capacity is called an *augmenting* path. Computing one is easy. First delete the *saturated* edges—those with $f[u, w] = \text{cap}(u, w)$. Then the possible augmenting paths (the s - t paths with positive residual capacity) are the s - t paths in the resulting graph. Such a path can be found (if it exists) in linear time using depth-first or breadth-first search.

The greedy algorithm maintains the invariant that f is a flow (that is, f satisfies the capacity and conservation constraints). The invariant holds initially because each edge has flow zero. In each iteration, in Step 4.2, increasing the flow by $\delta(p)$ along the s - t path p maintains the invariant, because it maintains the capacity constraints and the conservation constraints. (Verify!) So, if the algorithm terminates, it returns a flow. Unfortunately, the algorithm doesn't work—it can get stuck before it finds a maximum flow, as shown in Figure 8.1.

8.1.3 The Ford-Fulkerson algorithm

In the remainder of this note, we assume that in G no pair $u, w \in V$ of vertices has edges in both directions ($|\{(u, w), (w, u)\} \cap E| \leq 1$). This assumption is just to simplify the notation. It is easy to reduce the general case to this one by “splitting” edges as necessary.

Intuitively, in the example, the greedy algorithm failed because it pushed too much flow along edge (a, b) . To fix it, we introduce a mechanism for allowing the algorithm to “cancel” previously sent flow. Namely, for every edge $(u, w) \in E$, if the current flow f sends positive flow $f(u, w) > 0$ on the edge, we add an artificial *back edge* (w, u) , giving it capacity $f(u, w)$. We also remove all saturated edges. The resulting graph is the *residual graph* $G_f = (V, E_f)$ for f , where

$$E_f = \{(u, w) \in E : f(u, w) < \text{cap}(u, w)\} \cup \{(u, w) : (w, u) \in E, f(w, u) > 0\}.$$

The edge capacities in G_f , called *residual capacities* are

$$\text{cap}_f(u, w) = \begin{cases} \text{cap}(u, w) - f(u, w) & \text{if } (u, w) \in E \text{ and } f(u, w) < \text{cap}(u, w) \\ f(w, u) & \text{if } (w, u) \in E \text{ and } f(w, u) > 0. \end{cases}$$

Ford-Fulkerson ($G = (V, E), \text{cap}, s, t$)	
1. initialize $f[u, w] = 0$ for all $(u, w) \in E$	
2. for any s - t path p in G_f , define $\delta(p) = \min\{\text{cap}_f(u, w) : (u, w) \in p\}$	(p 's residual capacity)
3. while there is an s - t path in G_f :	
4.1. let p be any such path; let $\Delta = \delta(p)$	(note $\delta(p) > 0$ by def'n of G_f)
4.2. for each edge $(u, w) \in p$:	(augment flow along augmenting path p by Δ units)
4.3. if $(u, w) \in E$: increase $f[u, w]$ by Δ	
4.4. else: decrease $f[w, u]$ by Δ	(cancel Δ units of flow on the reverse edge (w, u))
5. return f	

Figure 8.2: The Ford-Fulkerson augmenting-paths algorithm for Maximum Flow.

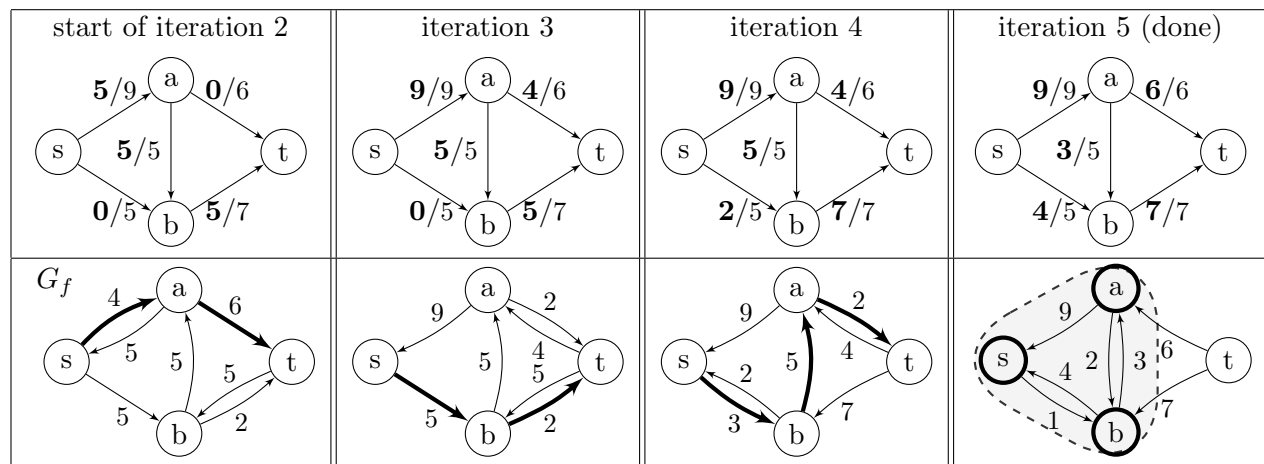


Figure 8.3: Example execution of the Ford-Fulkerson algorithm. The first row shows the flow f in the graph G at the start of each iteration (2–5). The second row shows the residual graph G_f and the augmenting path p in that iteration. After Iteration 5 there is no s - t path in the residual graph G_f (as there are no edges leaving the s - t cut with $S = \{s, a, b\}$), so the algorithm terminates.

We modify the greedy algorithm to maintain the residual graph G_f , and, in each iteration, to use any s - t path in G_f as the augmenting path. (Note that all residual capacities are positive.) This gives the Ford-Fulkerson algorithm, shown in Figure 8.2.

Observation 8.1. *The Ford-Fulkerson algorithm maintains the invariant that f is a flow in G .*

The algorithm maintains the invariant because, in each iteration, augmenting flow along the path p preserves the conservation and capacity constraints. (To see that this still holds when flow is “canceled” on an edge, try some examples.) Figure 8.3 shows the execution of the modified algorithm on the example from Figure 8.1.

Theorem 8.1 (Ford & Fulkerson). *If the algorithm terminates, it returns a maximum s - t flow f .*

Proof. Assume the algorithm terminates. Let f be the flow it returns. Let S contain the set of vertices reachable from the source s in the final residual graph G_f . Let $T = V \setminus S$. The sink t is not in S , because at termination there is no s - t path in the residual graph. So (S, T) is an s - t cut. (In Figure 8.4, $S = \{s, a, b\}$.)

We use the following observation, which follows from the definition of S : in G_f there is no edge (u, w) from S to T (that is, with $u \in S$ and $w \in T$). (Otherwise, since u is reachable from s in G_f , w would also be reachable from s , so would be in S .)

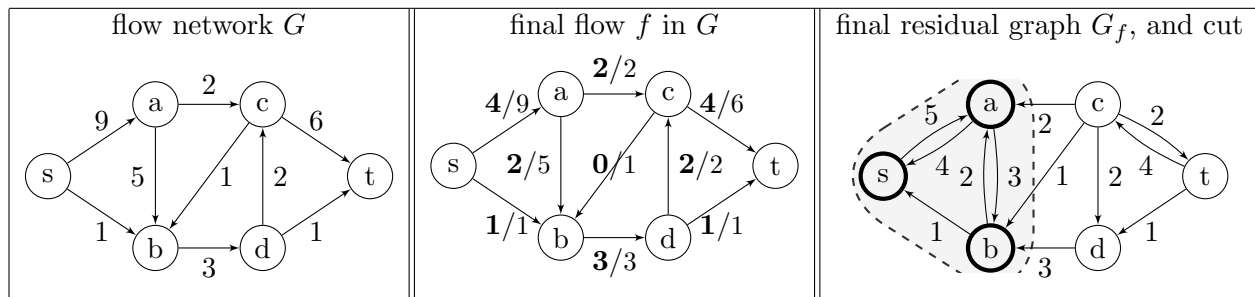


Figure 8.4: An example to illustrate the proof of Theorem 8.1.

Consider any edge $(u, w) \in E$ from S to T in the original graph G . By the observation above, this edge is not present in the residual graph G_f . This implies that edge (u, w) must be saturated: $f[u, w] = \text{cap}(u, w)$. Summing over all edges from S to T in E , we get $f(S, T) = \text{cap}(S, T)$. (In Figure 8.4, these edges are (a, c) and (b, d) .)

Consider any edge $(u, w) \in E$ from T to S in the original graph G . The reverse edge (w, u) (which goes from S to T) is not present in the residual graph (by the same observation). By definition of G_f , this implies that edge (u, w) has no flow: $f(u, w) = 0$. Summing over all edges from T to S in E , we get $f(T, S) = 0$. (In Figure 8.4, there is one such edge, (c, b) .)

By Lemma 8.1 and the previous two paragraphs, $|f| = f(S, T) - f(T, S) = f(S, T) = \text{cap}(S, T)$. That is, the value of f equals the capacity of the cut. By Lemma 8.2, no flow has value more than $\text{cap}(S, T)$. So f is a maximum flow. $\square$

Python code: [max_flow.py](#) (Section 8.4).

8.1.4 Consequences: The Max-Flow/Min-Cut Theorem; Integer Flows

We conclude with two important consequences.

Theorem 8.2 (max-flow/min-cut). *In any flow network, the maximum s - t flow value equals the minimum capacity of any s - t cut.*

Proof. Let v^* denote the maximum value of any s - t flow. (It is well defined, as the set of flows is closed under limit.) Let f be a flow of value v^* . Let S contain the vertices reachable from s in G_f . Then $t \notin S$: otherwise an s - t path in G_f would be an augmenting path, and augmenting f along it would give a flow of value more than v^* . Following the reasoning in the proof of Theorem 8.1, (S, T) is an s - t cut with capacity equal to $|f|$. By Lemma 8.2, every cut has capacity at least $|f| = v^*$. So $\text{cap}(S, T) = v^*$ is the minimum possible s - t cut capacity. $\square$

For many applications, the edge capacities are integers, and we want an *integer-valued* flow, where the flow $f(u, w)$ on each edge (u, w) is an integer.

Theorem 8.3. *If the edge capacities are integers, the algorithm returns an integer-valued flow.*

Proof. Assume the edge capacities are integers. The algorithm maintains the invariant that f is integer-valued, because if f is integer valued and each capacity is an integer, then each capacity in the residual graph is a positive integer, so $\delta(p)$ is a positive integer in each iteration. And the algorithm terminates, because in each iteration the value of the flow increases by at least 1, but the value never exceeds the capacity of any s - t cut. $\square$

Note that this algorithm may take exponentially long. Indeed, there are flow networks (with irrational edge capacities) in which the algorithm might not even terminate (depending on how augmenting paths are chosen)! But there are variants of the algorithm that are guaranteed to run in polynomial time. For example, the Edmonds-Karp algorithm (discovered independently by Dinitz) specializes the Ford-Fulkerson algorithm by choosing the augmenting path in each iteration to have a minimum number of edges. (Such a path can be found using breadth-first search.) With this choice, one can prove there are at most $O(|V| \cdot |E|)$ iterations, so the running time is $O(|V| \cdot |E|^2)$.

Python code: [max_flow.py](#) (Section 8.4).

8.1.5 Proof of Lemma 8.1

Fix any flow network $(G = (V, E), \text{cap}, s, t)$.

Lemma 8.1. *Let f and (S, T) be any flow and s - t cut. Then $|f| = f(S, T) - f(T, S)$.*

That is, the value of the flow equals the flow from S to T minus the flow from T to S .

Proof. Fix any such flow f and cost (S, T) . For ease of notation, if, for a pair of vertices $u, w \in V$ there is no edge $(u, w) \in E$, define $f(u, w) = 0$. Following this convention, for every vertex v other than s or t the conservation constraint for v is $\sum_{w \in V} f(v, w) = \sum_{u \in V} f(u, v)$. Then

$$\begin{aligned}
 |f| &= \sum_{w \in V} f(s, w) - \sum_{u \in V} f(u, s) && \text{(by definition of } |f| \text{)} \\
 &= \sum_{v \in S} \sum_{w \in V} f(v, w) - \sum_{v \in S} \sum_{u \in V} f(u, v) && \text{(adding all conservation constraints for } v \in S \setminus \{s\} \text{)} \\
 &= \sum_{v \in S} \sum_{w \in V \setminus S} f(v, w) - \sum_{v \in S} \sum_{u \in V \setminus S} f(u, v) && \text{(canceling the common terms for edges from } S \text{ to } S \text{)} \\
 &= \sum_{v \in S} \sum_{w \in T} f(v, w) - \sum_{v \in S} \sum_{u \in T} f(u, v) && \text{(because } T = V \setminus S \text{)} \\
 &= f(S, T) - f(T, S) && \text{(by definition of } f(S, T) \text{ and } f(T, S) \text{).}
 \end{aligned}$$

□

8.1.6 External sources on Max Flow

- [DPV](#) Chapter 7.2. [CLRS](#) Chapter 26. [KT](#) Chapter 7. [JE](#) Chapter 10. MIT lecture video:
 - <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-videos/lecture-13-incremental-improvement-max-flow-min-cut/>

8.2 Polynomial-time reductions

So far our primary strategy for solving a given computational problem has been to design an algorithm from scratch. To this end, we’ve studied basic algorithm-design techniques such as divide-and-conquer, dynamic programming, and greedy algorithms. In the remainder of these notes, we consider a different approach: solving the problem that you want to solve by *reducing* it to a different problem, one that already has a known algorithm. More broadly, our interest is in how different computational problems relate to each other. We start with some necessary formal definitions, then illustrate the concept by describing how to reduce some new problems to Max Flow.

Decision problems (definition). When studying reductions, for simplicity and for historical reasons, we usually cast the problems of interest as *decision problems*—problems with yes/no answers. That is, for any input, the output is either ‘yes’ or ‘no’. Most computational problems can easily be recast in this way. For example, here is the decision version of Max Flow:

Max Flow (decision version)

input: a flow network (digraph $G = (V, E)$ with edge capacities, source s and sink t), and a $k \geq 0$

output: ‘yes’ if G has an s - t flow with value at least k , and otherwise ‘no’

Polynomial-time (definition). The first question we consider when approaching a new computational problem is whether the problem has an algorithm that runs in *polynomial time*—that is, for every possible input, the algorithm computes a correct output in time bounded by some polynomial in the size of the input. The degree of the polynomial can be large, as long as it’s constant. (For example, $O(n^{1000})$ is polynomial time. $O(n \log n)$ is also polynomial time, as $O(n \log n) = O(n^2)$.)

Polynomial-time reductions (definition). Now we can formally define a *polynomial-time reduction*. Let A and B be any two decision problems. A *reduction* from A to B is a function r that maps each input x for A to a corresponding input, $r(x)$, for B . The reduction is *correct* if it has the following property:

For every possible input x to A ,
the answer to x is **yes** if and only if the answer to $r(x)$ (as an instance of B) is **yes**.

That is, if we knew the answer to $r(x)$, it would tell us the answer to x .

And the reduction r is a *polynomial time* reduction if there is an algorithm that, given any input x outputs $r(x)$, with worst-case run time bounded by a polynomial in the length of x . The notation

$$A \leq_P B$$

means “ A reduces in polynomial time to B ”. It means that (up to polynomial factors) A is “no harder” than B : if $A \leq_P B$, and B has a polynomial-time algorithm, then so does A . This is because composing the reduction with any polynomial-time algorithm for B gives a polynomial-time algorithm for A .

8.2.1 Example 1: Bipartite Matching $\leq_P$ Max Flow

To illustrate the concept, consider the following decision problem:

Bipartite Matching (decision version)
input: a bipartite graph $G = (U, W, E)$, and a value k
output: ‘yes’ if G has matching of size at least k , and otherwise ‘no’

Note that a bipartite graph $G = (U, W, E)$ is an undirected graph in which the vertices are divided into two parts (sets) U and W , and each edge (u, w) in E is in $U \times W$ (that is, u is in U and w is in W , so all edges go between the parts—no edge has both endpoints in one part).

[Consider an example for intuition.]

Next we define a polynomial-time reduction r from this problem to Max Flow. To define the reduction, one specifies, for every possible input I (for Bipartite Matching) what the output $r(I)$ is. The output must be some input for Max Flow. We have to explicitly define this input in terms of I (that is, from the bipartite graph G and k). Here is the definition of our reduction r :

Step 1. Define the reduction. Do this by specifying its output for every possible input.

Our reduction r from Bipartite Matching to Max Flow:

input: any Bipartite Matching instance $I = (G = (U, W, E), k)$
output: the output $r(I)$ is the Max Flow instance $(G' = (V', E'), s, t, k')$ where

$$\begin{aligned} k' &= k \\ V' &= U \cup W \cup \{s, t\} \\ E' &= E \cup \{(s, u) : u \in U\} \cup \{(w, t) : w \in W\} \end{aligned}$$

and all edges have capacity 1.

In words: The reduction takes $k' = k$, then obtains G' from G as follows. The vertices of G' are the vertices of G and two new vertices s and t . For each edge (u, w) from U to W in G , there is a directed edge (u, w) in G' with capacity 1. For each vertex $u \in U$, there is a directed edge (s, u) in G' with capacity 1. For each vertex $w \in W$, there is a directed edge (w, t) in G' with capacity 1.

[Consider an example for intuition. See also [KT Section 7.5](#), which presents the same reduction.]

Step 2. Explain how the reduction can be computed in polynomial time. To know that the reduction r is a polynomial-time reduction, we have to be sure that, given any input I , the output $r(I)$ can be computed in time polynomial in the size of I .

By inspection of the reduction r defined above, given G and k , G' (and k') can be computed in time linear in the size of G , by simply enumerating the vertices and edges of G appropriately. So the reduction can be computed in polynomial (even linear) time.

Step 3. Prove that the reduction is correct. To know that the reduction r is correct, we have to be sure that, for every possible input I , the answer to the output $r(I)$ is yes if and only if the answer to I is yes. (In other words, to solve I , it suffices to solve $r(I)$.)

Next we’ll give a detailed proof. Study the form of this proof carefully. The reasoning for the correctness of any reduction always has the same overall structure. Only the details vary, depending on the specific reduction.

Note: Don't be intimidated by the length of the proof. Most of the steps in the proof are straightforward. Make sure you understand the high-level structure of the reasoning. A short-form proof follows the long-form proof. You may wish to review it first.

Lemma 8.3. *The reduction r is correct. That is, for every instance I of Bipartite Matching, the answer to I is yes if and only if the answer to $r(I)$ (as an instance of Max Flow) is yes.*

Proof (long form).

1. Consider any possible input $I = (G = (U, W, E), k)$. *block: $\forall$*
2. Let $G' = (V', E')$ and k' be the output $r(I)$ of the reduction given I .
3. First, we'll prove that, if the answer to I is yes, then the answer to $r(I)$ is yes.
Then, we'll prove the converse: if the answer to $r(I)$ is yes, then the answer to I is yes.
- 4.1. Suppose that the answer to I is yes. *block: if ... then ...*
- 4.2. That is, G has some matching, say M , of size at least k .
- 4.3. We'll show that G' has some s - t flow of size at least $k' = k$.
- 4.4. That is, we'll show that the answer to $r(I)$ is yes.
- 4.5. Define flow f as follows:
for every edge (u, w) in M , let $f(s, u) = 1$, let $f(u, w) = 1$, and let $f(w, t) = 1$.
(That is, send 1 unit of flow along the path (s, u, w, t) .)
For every other edge (u, w) in G' , let $f(u, w) = 0$.
- 4.6. By inspection of f , the value of f is the size of M , which is at least $k' = k$.
- 4.7. And f satisfies the conservation and capacity constraints, so f is a valid flow.
- 4.8. So G' has an s - t flow of size at least k' .
- 4.9. So, the answer to $r(I)$ is yes.
5. By Block 4, if the answer to I is yes, then the answer to $r(I)$ is yes.
6. Next we show the converse.
- 7.1. Suppose that the answer to $r(I)$ is yes. *block: if ... then ...*
- 7.2. That is, G' has some s - t flow, of value at least $k' = k$.
- 7.3. Since G' has integer capacities, G' has such a flow for which each edge flow is an integer.
- 7.4. Let f be such a flow. Since each edge capacity in G' is 1, each edge flow $f(u, w)$ is either 0 or 1.
- 7.5. We'll show that G has some matching of size at least k .
- 7.6. Define $M = \{(u, w) \in E : f(u, w) = 1\}$ to contain each edge in G that has flow in G' .
- 7.7. For each vertex $u \in U$, there is only one edge into u in G' .
- 7.8. So, because of the conservation constraints, each vertex $u \in U$ is in at most one edge in M .
- 7.9. Likewise, each vertex $w \in W$ is in at most one edge in M .
- 7.10. So M is a matching in G .
- 7.11. Also by the conservation constraints, for each edge (s, u) in G' with $f(s, u) = 1$ there is some edge (u, w) in M , so the size of M is at least the value of f , that is, at least k .
- 7.12. So M is a matching of size at least k in G .
- 7.13. So the answer to I is yes.
8. By Block 7, if the answer to $r(I)$ is yes, then the answer to I is yes.
9. By this and Step 5, the answer to I is yes if and only if the answer to $r(I)$ is yes. □

As mentioned above, the high-level structure of the proof of correctness is the same for all reductions. Only the details (within the two blocks) depend on the specific reduction. For readers who understand this, it is enough to give just those details. We could do that as follows:

Proof of correctness, short form. Consider any possible input $I = (G = (U, W, E), k)$ to the reduction. Let $G' = (V', E')$ and $k' = k$ be the output of the reduction for that input.

Suppose that G has some matching M of size at least k . Define a flow f in G' by taking $f(s, u) = f(u, w) = f(w, t) = 1$ for every matched edge (u, w) , and giving zero flow to the remaining edges. Then f is an s - t flow whose value is equal to the size of M , which is at least k . So, if G has a matching of size at least k , then G' has a flow of value at least k' .

Conversely, suppose that G' has a flow of value at least $k' = k$. Let f be such a flow that is also integer-valued (there must be one, since the capacities in G' are integers). So, the flow on each edge is either 1 or 0. Let M contain the edges in E whose corresponding edges in G' have flow 1. For each vertex $u \in U$, there is only one edge into u in G' , so (by the conservation constraints), each vertex $u \in U$ is in at most one edge in M . Likewise, each vertex $w \in W$ is in at most one edge in M . So M is a matching in G . Since the value of f is at least k , there are at least k edges out of s that have flow 1. So, by inspection of G' (and the conservation constraints), there are at least k edges in M . So M is a matching of size at least k in G .

By the above reasoning, G has a matching of size at least k if and only if G' has an s - t flow of value at least k' . That is, the answer to I is **yes** if and only if the answer to $r(I)$ is **yes**. So the reduction is correct. $\square$

8.2.2 Example 2: Disjoint Paths $\leq_P$ Max Flow

We give one more reduction, from the following problem to Max Flow.

Disjoint Paths (decision version)

input: a digraph $G = (V, E)$, source s , sink t , and a value k

output: ‘yes’ if G there are k s - t paths in G such that no two share an edge; otherwise ‘no’.

[Consider an example for intuition.]

Next we define a polynomial-time reduction r from this problem to Max Flow. To define the reduction, we specify, for every possible input I (for Disjoint Paths) what the output $r(I)$ is. The output must be some input for Max Flow, that is, a flow network and a value k . We have to say specifically how to obtain this flow network from I (that is, from G , s , t , and k). Here is the definition of our reduction r :

Step 1. Define the reduction. (Specify the output for every possible input.)

Here is our reduction r from Disjoint Paths to Max Flow:

input: any Disjoint Paths instance $I = (G = (V, E), s, t, k)$

output: the output $r(I)$ is the Max Flow instance (G', s, t, k) where G' is obtained from G by assigning capacity 1 to each edge.

[Consider an example for intuition. See also [KT Section 7.6](#), which presents the same reduction.]

Step 2. Explain how the reduction can be computed in polynomial time. By inspection of the reduction r (as defined above), given $I = (G, s, t, k)$, the flow network G' can be computed in time linear in the size of G , by simply adding a capacity to each edge. So the reduction can be computed in polynomial (even linear) time.

Step 3. Prove that the reduction is correct. Following the previous reduction from Bipartite Matching, we'll assume the reader understands the general high-level structure common to all proofs of correctness for reductions. We'll give a short-form proof. Before that, we need a utility lemma.

Lemma 8.4. *Given any s - t flow f with positive value, t is reachable from s using only edges that have positive flow.*

Proof (short form). Let G'' be the flow network obtained by reducing the capacity of each edge to equal the flow on the edge, then deleting all edges that have flow (and modified capacity) zero. The original flow remains valid in G'' .

Suppose for contradiction that there is no s - t path in G'' . Let S denote the set of vertices reachable from s in G'' , so that $(S, T = V \setminus S)$ is an s - t cut. Note that no edge (u, w) in G'' leaves S (otherwise u would be reachable from s , but w would not be, even though there is an edge from u to w). So, (S, T) is an s - t cut of capacity zero in G'' . So, every s - t flow in G'' has value at most zero. This is a contradiction, as the original flow is valid in G'' and has positive value. $\square$

Here is the proof that the reduction is correct.

Lemma 8.5. *The reduction r is correct. That is, for every input I , the answer to I is yes if and only if the answer to $r(I)$ is yes.*

Proof (short form). Consider any possible input $I = (G = (V, E), s, t, k)$ to the reduction. Let G' (and s, t, k) be the output of the reduction for that input.

First suppose that G does have k edge-disjoint s - t paths, say, $P_1, P_2, \dots, P_k$. Assume without loss of generality that each path is simple. (If not, just remove all cycles from each path to get k simple edge-disjoint paths.) Define a flow f in G' by setting $f(u, w)$ to be the number of times (u, w) occurs on these paths. Since the paths are edge-disjoint and simple, each flow value $f(u, w)$ will be 0 or 1. Since each edge capacity in G' is 1, f obeys the capacity constraints. Since each of the paths leaves s , the value of f is k . For each vertex v (other than s or t), for each edge (u, v) on some path P_i entering v , the next edge (v, w) on P_i leaves v , so the total flow into v equals the total flow leaving v . So f obeys the conservation constraints. So f is a valid flow in G' , of value k .

Conversely, suppose that G' does have a flow of value at least k . All edge capacities are integers, so there is also an *integer* flow of value at least k . Let f be such a flow. Since each edge capacity is 1, each flow value $f(u, w)$ is 0 or 1. Convert f into a sequence of paths $P_1, P_2, \dots, P_\ell$ as follows:

1. for $i = 1, 2, \dots$, while the value of flow f is positive:
 - 2.1. let P_i be an s - t path that uses only edges (u, w) such that $f(u, w) = 1$
 - 2.2. for each edge $(u, w) \in P_i$, reduce the flow $f(u, w)$ on (u, w) by 1 (to 0)

In each iteration, the s - t path P_i must exist by Lemma 8.4. The iteration reduces the flow on each edge in P_i by 1 (to 0). This preserves the conservation constraints, the capacity constraints, and the property that the flow on each edge is either 0 or 1. But it reduces the value of the flow by 1. It follows that the number of iterations will equal the original value of f . Since this is at least k , the process produces at least k paths. And no edge occurs twice in these paths, because an edge (u, w) is used in a path P_i only when $f(u, w)$ is 1, and after that $f(u, w)$ is reduced to 0. So, if G' has a flow of value at least k , then G has k edge-disjoint s - t paths. (If the process produces more than k paths, just ignore some.) $\square$

8.2.3 Exercise: Bipartite Matching $\leq_P$ Disjoint Paths

As an exercise, find a poly-time reduction from Bipartite Matching to Disjoint Paths.

8.2.4 The two key properties of polynomial-time reductions

Recall that $A \leq_P B$ denotes that there is a polynomial-time reduction from decision problem A to decision problem B . Intuitively, this notation suggests that (up to polynomial time), A is “no harder” than B . Formally, the following holds:

Lemma 8.6. *Let A and B be decision problems such that A reduces in polynomial time to B . If B has a polynomial-time algorithm, then A has a polynomial-time algorithm. That is, if $A \leq_P B$ and $B \in P$, then $A \in P$.*

Proof. Assume r is a poly-time reduction from A to B , and that B has a poly-time algorithm. Fix c and c' such that r is computable in time $O(n^c)$ and B has an algorithm running in time $O(n^{c'})$. Consider the following algorithm for A :

algorithm for A (on input x):

1. compute $y = r(x)$, in time $O(|x|^c)$
2. run the algorithm for B on input y , taking time $O(|y|^{c'})$
3. if the algorithm for B outputs yes, then output yes, otherwise no

This algorithm for A is correct by the following reasoning:

The algorithm for A outputs yes on input x

$\iff$ the algorithm for B outputs yes on input y (by inspection of the algorithm for A)

$\iff$ the answer for $r(x)$ is yes (as the algorithm for B is correct)

$\iff$ the answer for x is yes (because r is a correct reduction from A to B)

So the algorithm is correct.

The algorithm takes polynomial time, $O(|x|^{c'})$, by the following reasoning. Step 1, computing $y = r(x)$, takes time $O(|x|^c)$ because $r(x)$ is computable in time $O(|x|^c)$. Also, this means that y has size $O(|x|^c)$. So Step 2, running the algorithm for B on y , takes time $O(|y|^{c'}) = O((|x|^c)^{c'}) = O(|x|^{c c'})$. $\square$

One more formal observation: poly-time reductions compose, so the relation $\leq_P$ is transitive:

Lemma 8.7. *Let A , B , and C be decision problems. If A reduces in polynomial time to B , and B reduces in polynomial time to C , then A reduces in polynomial time to C .*

That is, if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

Proof. Let r_1 and r_2 be the poly-time reductions from A to B , and from B to C . Consider the function r_3 defined by composing r_1 and r_2 , that is, $r_3(x) = r_2(r_1(x))$. The function r_3 is a correct reduction from A to C by the following reasoning:

The answer for x is yes

$\iff$ the answer for $r_1(x)$ is yes (because r_1 reduces A to B)

$\iff$ the answer for $r_2(r_1(x))$ is yes (because r_2 reduces B to C)

$\iff$ the answer for $r_3(x)$ is yes (because $r_3(x) = r_2(r_1(x))$)

So the reduction r_3 is correct.

For the running time, fix c_1 and c_2 such that r_1 is computable in time $O(n^{c_1})$ and r_2 is computable in time $O(n^{c_2})$. The following algorithm computes r_3 in polynomial time, $O(n^{c_1 c_2})$:

algorithm to compute $r_3(x)$:

1. compute $y = r_1(x)$, in time $O(|x|^{c_1})$
2. compute and return $r_2(y)$, taking time $O(|y|^{c_2})$

Clearly the algorithm computes $r_3(x)$. To bound the run time, reason as in the previous proof: Step 1 takes time $O(|x|^{c_1})$, so y has size $|y| = O(|x|^{c_1})$, so Step 2 takes time $O(|y|^{c_2}) = O(|x|^{c_1 c_2})$. $\square$

8.2.5 External sources on polynomial-time reductions

- Reducing matching to flow: [KT](#) Section 7.5; [DPV](#) Section 7.3; [CLRS](#) Section 26.3
- Reducing disjoint paths to flow: [KT](#) Section 7.6
- For Seam Carving (which reduces to shortest s - t path in a DAG), see
https://en.wikipedia.org/wiki/Seam_carving
<https://www.youtube.com/watch?v=6NcIJXTlugc>
- For Image Segmentation via Min Cut, see [KT](#) Section 7.10. See also:
<https://discovery.ucl.ac.uk/13383/1/13383.pdf>, Figures 1, 2, and 11
- Other reductions to Max Flow or Min Cut:
[KT](#) Sections 7.5–7.12 (and [KW slides](#) for Ch. 7); see also [JE](#) Chapter 11
- Reductions in general:
[KT](#) Section 8.1 (and [KW slides](#) for Ch. 8); [CLRS](#) Section 34.3; and [JE](#) Chapter 12, and
<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-23-computational-complexity/>

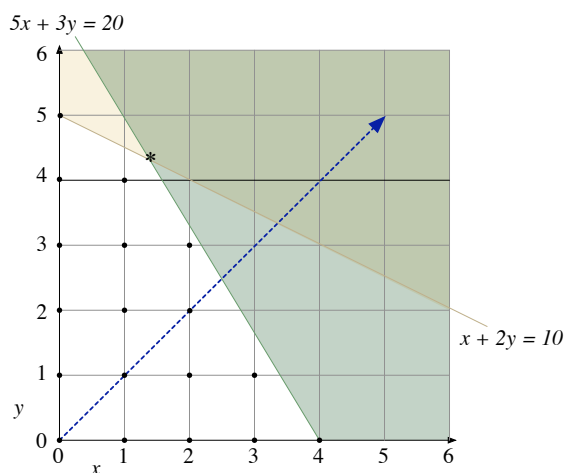


Figure 8.5: The two-dimensional linear program $\max\{x + y : x, y \geq 0; x + 2y \leq 10; 5x + 3y \leq 20\}$. The feasible region is the white portion of the positive quadrant. The optimal point is marked with an asterisk.

8.3 Linear Programming and Integer Linear Programming

Recall that linear algebra is the study of systems of linear equations, which are useful for modeling a wide variety of problems. *Linear programming* can be understood as a generalization of linear algebra, in that it is about systems of linear *inequalities* (including equations). Linear *inequalities* allow us to model an even wider variety of problems.

As a computational problem, an instance of Linear Programming requires finding a vector that optimizes a given linear objective function subject to given linear constraints. For example, given a matrix $A \in \mathbb{R}^{n \times m}$ and vectors $b \in \mathbb{R}^n$, $c \in \mathbb{R}^m$, find

$$\min\{c \cdot x \mid x \in \mathbb{R}^m; x \geq 0; Ax \geq b\}.$$

A vector inequality such as $x \geq 0$ is interpreted component-wise, i.e., as $x_i \geq 0$ for all $i \in \{1, 2, \dots, m\}$. Similarly, $Ax \geq b$ is interpreted component-wise.

A *feasible solution* is any vector x satisfying the constraints $x \in \mathbb{R}^m$, $x \geq 0$, and $Ax \geq b$. The objective function (or *cost*) is $c \cdot x = \sum_j c_j x_j$. Linear programs may take other forms, for example $\max\{c \cdot x \mid x \in \mathbb{R}^m; x \geq 0; Ax \leq b\}$.

Integer Linear Programming. *Integer Linear Programming* extends Linear Programming by allowing an additional constraint on the solution, namely, that each variable should be assigned an *integer* value. For example, given (A, b, c) as above, find $\min\{c \cdot x \mid x \in \mathbb{Z}^m; x \geq 0; Ax \geq b\}$. (Note that $\mathbb{Z}^m$ is the set of m -dimensional integer vectors.) Integer Linear Programming is a so-called *NP-hard* problem—it is powerful enough to model many computationally *intractable* combinatorial problems. So Integer Linear Programming is unlikely to have a polynomial-time algorithm. Nonetheless, many moderately sized Integer Linear Programs can be solved in practice, so formulating a problem as an integer linear program can be an effective way to solve it.

A small linear program. Consider the problem maximize $x + y$ subject to $x + 2y \leq 10$, $5x + 3y \leq 20$, and $x, y \geq 0$. The problem is illustrated in Figure 8.5. The feasible region is the white area in the positive quadrant. The dashed vector points in the direction of increasing cost ($x + y$). The optimal feasible solution is $(1\frac{3}{7}, 4\frac{2}{7})$ (marked by “*”), of cost $5\frac{5}{7}$. The feasible integer solutions

(feasible solutions of the corresponding integer linear program) are marked with black dots. Three of these have maximum cost 5. The solution $(2, 2)$ of cost 4 is a $1\frac{1}{4}$ -approximate solution to the integer linear program and a $1\frac{3}{7}$ -approximate solution to the linear program.

Algorithms for Linear Programming. Linear Programming is efficiently solvable. That is, there are polynomial-time algorithms that, given any linear program, find an optimal solution. Historically the so-called *Simplex algorithm* is the most common. Its worst-case run time is not polynomial, but it usually works quickly in practice. In recent decades *Interior-point* algorithms have superseded the Simplex algorithm. These are polynomial-time algorithms. The *Ellipsoid* algorithm was the first polynomial-time algorithm to be discovered, but it is slow in practice, so is of mostly theoretical interest.

Algorithms via Linear Programming. Here's one technique for solving a given computational problem:

- (i) Model the problem as a linear program, that is, reduce the problem to Linear Programming.
- (ii) Use an existing LP solver to solve the linear program, and, therefore, your problem.

This approach can be easier than designing an algorithm from scratch. In the rest of this section we describe how various problems reduce to Linear Programming.

We note in passing that there is another, more sophisticated, approach:

- (i) Model the problem as a linear program.
- (ii) Design an efficient algorithm that mimics what the Simplex algorithm does for that linear program.

For example, the Ford-Fulkerson Max-Flow algorithm and Dijkstra's algorithm for Single-Source Shortest Paths can be viewed as special cases of the Simplex algorithm, obtained by applying the Simplex algorithm (with suitable choices of pivots) to the linear programs for those problems. We won't discuss this further here, as we don't discuss how the Simplex algorithm works. Chapter 7 of [DPV](#) discusses this approach specifically for Max Flow.

8.3.1 Max Flow as a linear program

Linear programs are often given by explicitly writing the constraints, rather than using a matrix. For example, a given Max Flow instance $(G = (V, E), s, t)$ translates directly to the following linear program:

$$\begin{aligned} &\text{maximize} && \sum_{w \in \text{out}(s)} f_{sw} - \sum_{u \in \text{in}(s)} f_{us} && \text{subject to:} \\ &(\forall (u, w) \in E) && 0 \leq f_{uw} \leq \text{cap}(u, w) && \text{--- capacity constraints} \\ &(\forall v \in V \setminus \{s, t\}) && \sum_{w \in \text{out}(v)} f_{vw} = \sum_{u \in \text{in}(v)} f_{uv} && \text{--- conservation constraints} \end{aligned}$$

This is a (very easy!) polynomial-time reduction from Max Flow to Linear Programming.

Recall that, if the edge capacities are integers, then there is always an optimal integer-valued flow. So, in this case, this linear program always has an optimal integer solution, and finding one is not hard.

Recall that Bipartite Matching and Disjoint Paths reduce in polynomial time to Max Flow. Since Max Flow reduces to Linear Programming, it follows that these two problems also reduce to Linear Programming. As an exercise, you could try formulating each of these problems as a linear program directly.

8.3.2 Min Cut as a linear program

A given Min Cut instance $(G = (V, E), s, t)$ is equivalent to the following linear program:

$$\begin{aligned} & \text{minimize} && \sum_{(u,w) \in E} \text{cap}(u,w) x_{uw} && \text{subject to:} \\ & && y_s = 0; y_t = 1 \\ & (\forall v \in V) && 0 \leq y_v \leq 1 \\ & (\forall (u,w) \in E) && 0 \leq x_{uw} \\ & (\forall (u,w) \in E) && y_w - y_u \leq x_{uw} \end{aligned}$$

The next lemma summarizes the relation to Min Cut:

Lemma 8.8. *For any Min Cut instance, every s - t cut (S, T) yields an integer solution to this linear program whose cost $\sum_{(u,w) \in E} \text{cap}(u,w) x_{uw}$ equals $\text{cap}(S, T)$, and every integer solution yields an s - t cut whose capacity is at most the cost of the solution. In particular, the minimum cost of any integer solution is the minimum capacity of any s - t cut.*

Proof sketch. Given any s - t cut (S, T) , the corresponding (integer) solution (x, y) to the linear program is obtained by taking $y_v = 0$ for $v \in S$ and $y_v = 1$ for $v \in T$, and then $x_{uw} = 1$ for each edge (u, w) that crosses the cut, and $x_{uw} = 0$ for each edge that doesn't cross the cut. One can verify that this must be a feasible solution with the correct cost. Conversely, given any integer solution (x, y) to the linear program, the corresponding s - t cut (S, T) is obtained by taking $S = \{v \in V : y_v = 0\}$ and $T = \{v \in V : y_v = 1\} = V \setminus S$. This is an s - t cut, and its capacity is at most the cost of (x, y) : each edge (u, w) that crosses the cut has $y_w - y_u = 1$, so $x_{uw} \geq 1$. $\square$

Also, there is always an optimal integer solution to this linear program:

Lemma 8.9. *For any Min Cut instance, there is an optimal integer solution to this linear program.*

Proof. Here's a clever proof using [the probabilistic method](#).² Fix an instance and any optimal LP solution (x, y) . Choose a number $r \in [0, 1]$ uniformly at random, then let

$$S = \{v \in V : y_v \leq r\} \text{ and } T = \{v \in V : y_v > r\}.$$

This gives a random s - t cut (S, T) . By [linearity of expectation](#), the expected capacity of this cut is

$$\begin{aligned} \sum_{(u,w) \in E} \text{cap}(u,w) \Pr[(u,w) \text{ crosses the cut}] &= \sum_{(u,w) \in E} \text{cap}(u,w) \Pr[y_u \leq r < y_w] \quad (\text{by definition of } (S, T)) \\ &= \sum_{(u,w) \in E} \text{cap}(u,w) \max(0, y_w - y_u) \quad (\text{by definition of } r) \\ &\leq \sum_{(u,w) \in E} \text{cap}(u,w) x_{uw} \quad (\text{by inspection of the LP}) \end{aligned}$$

²If you're not familiar with probability, this proof won't make much sense.

That is, the expected capacity of the random cut (S, T) is at most the cost of (x, y) . Thus, there must *exist* a cut (S^*, T^*) whose capacity is at most the cost of (x, y) . By Lemma 8.8, this cut (S^*, T^*) has a corresponding integer solution (x^*, y^*) whose cost equals the capacity of (S^*, T^*) , which is in turn at most the cost of (x, y) . So (x^*, y^*) is also optimal. $\square$

8.3.3 Shortest s - t path as a linear program

Given a directed graph $G = (V, E)$ with non-negative edge weights, and two vertices s and t , consider problem of finding a shortest path from s to t in G . Here are two linear programs that model this problem. We call one the *primal* and the other the *dual*:³

<i>primal</i>	<i>dual</i>
minimize $\sum_{(u,w) \in E} \text{wt}(u,w) f_{uw}$ subject to	maximize $\pi_t - \pi_s$ subject to
$(\forall v \in V) \quad \sum_{w \in \text{out}(v)} f_{vw} - \sum_{w \in \text{in}(v)} f_{wv} = \begin{cases} 1 & \text{if } v = s \\ -1 & \text{if } v = t \\ 0 & \text{otherwise} \end{cases}$	$(\forall (u,w) \in E) \quad \pi_w - \pi_u \leq \text{wt}(u,w)$
$(\forall (u,w) \in E) \quad f_{uw} \geq 0$	

For intuition, note that, in the primal linear program, f is a flow of value 1 from s to t . Integer flows of this kind correspond to s - t paths:

Lemma 8.10. *Every s - t path P in G yields an integer solution f to the primal whose cost is $\text{wt}(P)$, and every integer solution f yields an s - t path whose weight is at most the cost of f . In particular, the minimum cost of any integer solution is $\text{dist}(s, t)$.*

Proof sketch. For any s - t path P , the corresponding solution f is defined by giving f_{uw} a value equal to the number of times edge (u, w) occurs in P . Because the path starts at s , ends at t , and enters each other vertex v the same number of times it leaves v , this assignment to f meets the constraints of the primal LP. By inspection the cost of f in the LP equals $\text{wt}(P)$.

Conversely, given any integer solution f to the LP, one can construct a corresponding path P by starting at s , then walking greedily, taking each step along any edge (u, w) where f_{uw} is positive. When traversing an edge (u, w) , append the edge to P and decrement f_{uw} . Because f satisfies the LP constraints, a greedy step is always possible until the walk reaches t . At that point P is an s - t walk of weight at most the cost of f (the flow left over on edges of f costs at least 0), and, because the edge weights are non-negative, P contains an s - t path of weight at most $\text{wt}(P)$. $\square$

Lemma 8.11. *The primal LP always has an optimal integer solution.*

Proof sketch. By an argument similar to the greedy construction in the previous proof, one can show that any solution f can be expressed as a weighted average of some integer solutions $f_1, f_2, \dots, f_\ell$. The cost of f equals the average cost of these solutions, so the cheapest of these integer solutions has cost at most the cost of f . $\square$

It follows that the value of the linear program (that is, the optimal objective value achieved by any feasible solution) is $\text{dist}(s, t)$. Next we consider the dual LP, and show the same for it:

³The existence of two such linear programs for a problem is not a coincidence. It happens for every linear-programming problem! The Max-Flow/Min-Cut relationship is another example.

Lemma 8.12. *The value of the dual LP is $\text{dist}(s, t)$.*

Proof. First we show that the value of the dual LP is at most $\text{dist}(s, t)$. Let π^* be an optimal solution to the dual LP, so $\pi_t^* - \pi_s^*$ is the value of the dual LP. Let P^* be a shortest s - t path, so $\text{wt}(P^*) = \text{dist}(s, t)$. Then, summing the dual LP constraints over edges in P^* gives

$$\text{dist}(s, t) = \text{wt}(P^*) = \sum_{(u,w) \in P^*} \text{wt}(u, w) \geq \sum_{(u,w) \in P^*} \pi_w^* - \pi_u^* = \pi_t^* - \pi_s^*.$$

The last step follows because the sum “telescopes”—for each vertex v on the interior of the path P^* , the term π_v^* occurs once with positive sign and once with negative sign, and these terms cancel each other.

To finish we show that the value of the dual LP is at least $\text{dist}(s, t)$. Define a dual solution π by taking $\pi_v = \text{dist}(s, v)$ for every v (assuming every vertex is reachable from s ; deleting the unreachable vertices changes neither $\text{dist}(s, t)$ nor the LPs’ values). This solution π is feasible, because by the triangle inequality,⁴ for any edge (u, w) ,

$$\pi_w - \pi_u = \text{dist}(s, w) - \text{dist}(s, u) \leq \text{wt}(u, w).$$

And the cost of π is $\pi_t - \pi_s = \text{dist}(s, t) - \text{dist}(s, s) = \text{dist}(s, t)$. So, the dual LP has a feasible solution of cost $\text{dist}(s, t)$, and is a maximization problem, so its value must be at least $\text{dist}(s, t)$. $\square$

8.3.4 Subset Sum as an integer linear program

For variety, consider the Smallest Subset Sum problem. Fix an instance $(v = (v_1, v_2, \dots, v_n), T)$. The problem is to determine the smallest subsequence of v that sums to T . Here is an equivalent instance of Integer Linear Programming:

$$\begin{aligned} &\text{minimize} \quad \sum_{i=1}^n x_i \quad \text{subject to:} \\ &(\forall i \in \{1, \dots, n\}) \quad 0 \leq x_i \leq 1, \\ &\quad \quad \quad \sum_{i=1}^n v_i x_i = T, \\ &\quad \quad \quad x \in \mathbb{Z}^n. \end{aligned}$$

Setting $x_i = 1$ corresponds to including v_i in the subsequence. (And setting $x_i = 0$ corresponds to omitting it.) In this way, the solutions to this integer linear program correspond bijectively to the subsequences of v that sum to T . The size of each subsequence equals the cost, $\sum_i x_i$, of its corresponding solution x .

What would happen if we were to drop the explicit integrality constraint, “ $x \in \mathbb{Z}^n$ ”? This would give a linear program instead of an integer linear program. But that linear program wouldn’t necessarily have optimal solutions that are integer valued. Consider for example the Smallest Subset Sum instance $(v = (1, 3), T = 2)$. It has no solution—no subsequence of v sums to 2. But the corresponding linear program has many solutions, the optimum being $x = (0, 2/3)$.

8.3.5 External sources on linear programming

- [DPV](#) Chapter 7. [KT](#) Section 11.6 and [KW](#) slides. [CLRS](#) Chapter 29. MIT lecture video: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-videos/lecture-15-linear-programming-lp-reductions-simplex/>

⁴The triangle inequality is that $\text{dist}(s, w) \leq \text{dist}(s, u) + \text{wt}(u, w)$. It holds because one way to get from s to w is to take a shortest path from s to u , followed by the edge (u, w) .

- JE notes: <https://jeffe.cs.illinois.edu/teaching/algorithms/notes/H-lp.pdf>
- Microsoft Excel has a free add-on for solving LPs in a spreadsheet. See e.g. <https://www.fm-magazine.com/issues/2019/feb/linear-programming-microsoft-excel.html>
- For more serious applications, Gurobi has a commercial LP solver, free for academic use, that you can install on your own computer. It can also solve Integer Linear Programs of moderate size. <https://gurobi.com>
- Many other free LP solvers are available.

8.4 Python code

This section lists the Python code for this lecture note. Click on a file's name to open the file on GitHub. Copying code from GitHub is more reliable than copying it from this PDF, since some PDF viewers drop the leading spaces (the indentation) of copied lines.

[graph_library.py](#)

```
#!/usr/bin/env python3

from collections.abc import Callable, Hashable, Iterable, Iterator
from typing import Any

def out_neighbor_lists[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> dict[T, list[T]]:
    out_neighbors: dict[T, list[T]] = {v: [] for v in V}
    for u, w in E:
        out_neighbors[u].append(w)
    return out_neighbors

def in_neighbor_lists[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> dict[T, list[T]]:
    return out_neighbor_lists(V, [(w, u) for u, w in E])

def random_digraph(
    n: int = 10, m: int | None = None, acyclic: bool = False
) -> tuple[list[int], list[tuple[int, int]]:
    """
    return (vertex list, edge set) for random graph
    with n vertices and m edges
    """

    import random

    vertices = list(range(n))

    n_pairs = n * (n - 1)

    if m is None:
        m = int(n_pairs / 2)
    else:
        assert 0 <= m <= n_pairs

    def random_edge_set(n_edges: int) -> set[tuple[int, int]]:
        if acyclic:
            priority = random.sample(range(n), n)

        edges: set[tuple[int, int]] = set()
        while len(edges) < n_edges:
            u, w = random.choice(vertices), random.choice(vertices)
            if u != w:
                if acyclic and priority[u] < priority[w]:
                    u, w = w, u
                edges.add((u, w))
        return edges

    if m <= n_pairs / 2:
        edges = random_edge_set(m)
```

```

    else:
        # if most edges are present, faster to generate the complement
        all_pairs = set((u, w) for u in vertices for w in vertices if u != w)
        assert len(all_pairs) == n_pairs
        edges = all_pairs - random_edge_set(n_pairs - m)

    assert len(vertices) == n
    assert len(edges) == m
    return vertices, list(edges)

def random_graph(n: int = 10, m: int | None = None) -> tuple[list[int], list[tuple[int, int]]]:
    V, E = random_digraph(n, m, acyclic=True)
    E += [(w, u) for u, w in E]
    return V, E

def random_weighted_digraph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[tuple[int, int]], dict[tuple[int, int], int]]:
    import random

    V, E = random_digraph(*args, **kwargs)
    weights = {e: random.choice(range(10000)) for e in E}
    return V, E, weights

def random_weighted_graph(
    *args: Any, **kwargs: Any
) -> tuple[list[int], list[tuple[int, int]], dict[tuple[int, int], int]]:
    V, E, weights = random_weighted_digraph(*args, **kwargs, acyclic=True)
    E += [(w, u) for u, w in E]
    for u, w in E:
        weights[(w, u)] = weights[(u, w)]
    return V, E, weights

type Visit[T] = Callable[[T, T | None], object]

def depth_first_search[T: Hashable](
    V: Iterable[T],
    E: Iterable[tuple[T, T]],
    previsit: Visit[T] | None = None,
    postvisit: Visit[T] | None = None,
    source: T | None = None,
) -> None:
    neighbors = out_neighbor_lists(V, E)
    visited: set[T] = set()

    def DFS1(u: T, parent: T | None = None) -> None:
        visited.add(u)
        if previsit:
            previsit(u, parent)
        for w in neighbors[u]:
            if w not in visited:
                DFS1(w, u)
        if postvisit:
            postvisit(u, parent)

    if source is not None:

```

```

        DFS1(source)
    else:
        for v in V:
            if v not in visited:
                DFS1(v)

def breadth_first_search_parents[T: Hashable](
    V: Iterable[T], E: Iterable[tuple[T, T]], source: T | None = None
) -> Iterator[tuple[T, T]]:
    from collections import deque

    if source is None:
        source = next(v for v in V)

    neighbors = out_neighbor_lists(V, E)

    queue: deque[T] = deque()
    queue.append(source)
    visited = {source}

    while queue:
        u = queue.popleft()
        for w in neighbors[u]:
            if w not in visited:
                visited.add(w)
                queue.append(w)
                yield (w, u)

    # for v in V:
    #     if v not in d:
    #         d[v] = inf

    # return d, parents

def topological_sort[T: Hashable](V: Iterable[T], E: Iterable[tuple[T, T]]) -> list[T]:
    finished: list[T] = []

    def postvisit(u: T, parent: T | None) -> None:
        finished.append(u)

    depth_first_search(V, E, postvisit=postvisit)

    return list(reversed(finished))

```

max_flow.py

```

#!/usr/bin/env python3

from collections.abc import Hashable, Mapping, Set
from typing import Any

from graph_library import (
    breadth_first_search_parents,
    depth_first_search,
    in_neighbor_lists,
    out_neighbor_lists,
)

```

```

def Ford_Fulkerson[T: Hashable](
    V: Set[T],
    E: Set[tuple[T, T]],
    caps: Mapping[tuple[T, T], int],
    s: T,
    t: T,
    use_BFS: bool = False,
) -> dict[tuple[T, T], int]:
    assert all((w, u) not in E for (u, w) in E)

    f = {(u, w): 0 for (u, w) in E}

    def cap(u: T, w: T) -> int:
        return caps[u, w]

    def residual_cap(u: T, w: T) -> int:
        return cap(u, w) - f[u, w] if (u, w) in E else f[w, u]

    while True:
        Ef = set((u, w) for (u, w) in E if f[u, w] < cap(u, w)) | set(
            (w, u) for (u, w) in E if f[u, w] > 0
        )

        if not use_BFS:
            parents: dict[T, T | None] = {}

            def previsit(w: T, u: T | None) -> None:
                if u is not None:
                    parents[w] = u # noqa: B023 (previsit is used only in this iteration)

            depth_first_search(V, Ef, previsit=previsit, source=s)
        else:
            parents = dict(breadth_first_search_parents(V, Ef, s))

        if t not in parents:
            break

        path: list[Any] = [t] # a list of vertices, then (below) of edges
        while path[-1] in parents:
            path.append(parents[path[-1]])
        assert path[-1] == s

        path = list(reversed(path))
        path = list(zip(path, path[1:])) # list of vertices -> list of edges

        delta = min(residual_cap(u, w) for (u, w) in path)
        for u, w in path:
            if (u, w) in E:
                f[u, w] += delta
            else:
                f[w, u] -= delta

        assert delta > 0

    # done. verify then return f

    # verify flow value is cut capacity

    in_neighbors = in_neighbor_lists(V, E)
    out_neighbors = out_neighbor_lists(V, E)
    
```

```

    parents[s] = None

    value = sum(f[s, w] for w in out_neighbors[s])
    cut_capacity = sum(cap(u, w) for (u, w) in E if u in parents and w not in parents)
    assert value == cut_capacity

    # verify capacity constraints

    assert all(0 <= f[u, w] <= cap(u, w) for u, w in E)

    # verify conservation constraints

    assert all(
        sum(f[u, v] for u in in_neighbors[v]) == sum(f[v, w] for w in out_neighbors[v])
        for v in V - {s, t}
    )
    return f

def Edmonds_Karp[T: Hashable](
    V: Set[T], E: Set[tuple[T, T]], capacities: Mapping[tuple[T, T], int], s: T, t: T
) -> dict[tuple[T, T], int]:
    return Ford_Fulkerson(V, E, capacities, s, t, use_BFS=True)

def test_max_flow() -> None:
    s, a, b, t = "sabt"
    V = {s, a, b, t}
    capacities = {(s, a): 9, (s, b): 5, (a, b): 5, (a, t): 6, (b, t): 7}
    E = capacities.keys()

    f1 = Ford_Fulkerson(V, E, capacities, s, t)
    f2 = Edmonds_Karp(V, E, capacities, s, t)
    value1 = sum(f1[s, w] for (u, w) in E if u == s)
    value2 = sum(f2[s, w] for (u, w) in E if u == s)

    assert value1 == value2 == 13

if __name__ == "__main__":
    test_max_flow()

    print("test passed")

```