Turing Reducibility

Last Updated: November 6th, 2023

1 Introduction

A common technique in everyday problem solving is to leverage the solution to one problem in order to solve another. For example, consider our friend Sam who must solve the problem of earning enough money to help support his way through college. One possible solution that Sam has considered is to work part-time delivering food for DoorDash. This solution would leverage the fact that he's already solved the problem of driving a vehicle from any one city location to another. Sam will likely have to solve tens of transport problems during a single work shift. In computer science, when an algorithm solves an instance of problem A by making one or more invocations to another algorithm that solves some problem B, then we say that A is *Turing reducible* to B, named after the British mathematician Alan Turing (1912-1954) who provided one of the earliest known theoretical models of computation that is functionally equivalent to the computers we use today (so long as our computers are idealized as having an infinite supply of memory). In this lecture we take a closer look at Turing reducibility and how it can be used as a means for devising algorithms.

2 Turing Reducibility

An important tool for desigining an algorithm to solve a problem A is to leverage an existing algorithm that solves another problem B by calling that algorithm one or more times for different instances of B in order to solve a single instance of A.

Example 2.1. Consider the problem of multiplying two positive numbers, say 5 and 3. Marcia is still learning the multiplication table, but she performs well in addition, and also knows that 5×3 means $(5 + 5) + 5$. Thus, she first solves the addition problem $5 + 5$ and gets the answer 10. She then solves the final addition problem, $10 + 5$ to obtain the answer of 15.

The following function returns the product of its two inputs by making calls to an `add` function that returns the sum of its two inputs. It essentially generalizes Marcia's solving method.

```
unsigned int multiply(unsigned int a, unsigned int b)
{
    int sum = 0;
    int i;

    for(i=1; i <= b; i = add(i,1))
        sum = add(sum,a);

    return sum;
}
```

In Example 2.1, Marcia *reduced* the `Multiply` problem to the `Add` problem. In other words, she devised an algorithm for multiplying two numbers that relies on solving one or more addition problems. Moreover, whenever the answer to an instance of `Add` is being sought to help solve an instance of `Multiply`, then we say that `Multiply` is making a **query** to the `Add`-**oracle**, i.e., an entity that is capable of providing solutions to instances of `Add`. The answer provided by the oracle is called a **query answer**. Note that the algorithm that is making queries to an oracle does not necessarily need to know how the oracle is providing its answers. In case of the `multiply` function in Example 2.1, the function just assumes that each `add` query will be correctly answered with no concern about how the answer is obtained. In fact, the oracle may provide answers to instances of a problem for which it is impossible to devise an algorithm for solving it.

Definition 2.2. Problem A is **Turing reducible** to problem B, denoted $A \leq_{\mathrm{T}} B$, iff there is some algorithm that can solve any instance x of A, and is allowed to make zero or more queries to a B-oracle, i.e. an oracle that provides solutions to instances of B.

Definition 2.3. If $A \leq_{\mathrm{T}} B$ via an algorithm whose running time $\mathrm{O}(n^k)$, for some $k > 0$, then A is said to be **polynomial-time Turing reducible** to B, denoted $A \leq_{\mathrm{T}}^{\mathrm{p}} B$. Note: this definition assumes that each B-query is answered in one step.

Interesting fact: the term *oracle* comes from ancient Greece, where the "oracle at Delphi" meant a high priestess who resided in a sanctuary located on Mt. Parnassus, and gave predictions and advice to both statesmen and citizens.

3 The `Reachability` problem

In the remainder of this lecture we provide two examples of polynomial-time Turing reducibility from both the `2SAT` and `Max Flow` problems to the `Reachability` decision problem.

Definition 3.1. An instance of the `Reachability` problem is a graph $G = (V, E)$ and vertices $u, v \in V$, and the problem is decide if there is a path in G from u to v.

The following algorithm decides `Reachability` and has a linear running time equal to $O(m + n)$, where $m = |E|$ and $n = |V|$.

Reachability Algorithm

Input: $G = (V, E)$, $u, v \in V$.

Output: `true` iff there is a path from u to v.

If $u = v$, then return `true`.

Initialize FIFO queue Q with u: $Q \leftarrow (u)$.

Mark u as having been reached.

While $Q \neq (\ )$

Remove vertex w from the front of Q: $Q \leftarrow Q - Q[0]$.

For each edge $(w, x) \in E$

If x is unmarked, then mark x and enter x into Q: $Q \leftarrow Q + (x)$.

If v is marked, then return `true`.

Return `false`.

Theorem 3.2. The Reachability Algorithm is correct and has the stated running time equal to $O(m+n)$.

Proof. We claim that, for all $i \geq 0$, if there is a path from u to x having length i, then x gets marked during the algorithm.

Basis step. Assume $i = 0$. Then necessarily $x = u$ which gets marked before entering the `while` loop.

Inductive step. Assume the claim is true for some $i \geq 0$. Consider a path from u to x having length $i+1$. Let w be the vertex that immediately precedes x in the path. Then there is a path from u to w having length i. By the inductive assumption, vertex w gets marked and added to Q. Thus, there will be a step in the algorithm where w is removed from Q and edge $(w, x) \in E$ will be examined. At this point x gets marked in case it has yet to be marked.

The above inductive proof shows that, if v is reachable from u, then the algorithm returns `true`, since there is a path from u to v. Conversely, we leave it as an exercise to prove that, if a vertex gets marked during the algorithm, then that vertex must be reachable from u (hint: use induction).

Running Time. To see that the algorithm runs in linear time, notice that the `while` loop requires at most n iterations and, assuming an undirected graph, each edge (w, x) in G must be considered at most twice: once if w is removed from Q, and a second time if x is removed from Q. Thus, the total number of steps equals $O(2m+n) = O(m+n)$. □

Example 3.3. Show the contents of the queue Q during the execution of the above algorithm on the graph $G = (V, E)$, where

$$V = \{a, b, c, d, e, f, g, h\}$$

and the edges are given by

$$E = \{(a, b), (a, c), (b, c), (b, d), (b, e), (b, g), (c, g), (c, f),$$

$$(d, f), (f, g), (f, h), (g, h)\}.$$

Decide if h is reachable from a.

4 Boolean Variable Assignments

Before introducing the `2SAT` decision problem, we need to understand the concept of a Boolean variable assignment.

Boolean Variable A variable is said to be **Boolean** iff its domain equal $\{0, 1\}$. We use lowercase letters, such as $x, y, z, x_1, x_2, \ldots$, etc., to denote a Boolean variable.

Assignment An **assignment** over a Boolean-variable set V is a function $\alpha : V \to \{0, 1\}$ that assigns to each variable $x \in V$ a value in $\{0, 1\}$. We may represent α using function notation, or as a labeled tuple.

Example: for the assignment α that assigns 1 to both x_1 and x_2, and 0 to x_3, we may use function notation and write $\alpha(x_1) = 1$, $\alpha(x_2) = 1$, and $\alpha(x_3) = 0$, or we may use tuple notation and write

$$\alpha = (x_1 = 1, x_2 = 1, x_3 = 0),$$

or

$$\alpha = (1, 1, 0),$$

if the associated variables are understood.

Variable Negation If x is a variable, then $\overline{x}$ is called its **negation**.

Example: Suppose assignment α satisfies $\alpha(x_1) = 0$. Then (extending α to include negation inputs) $\alpha(\overline{x}_1) = 1$.

Literal A **literal** is either a variable or the negation of a variable .

Example: x_1, x_3, $\overline{x}_3$, are $\overline{x}_5$ all examples of literals.

Consistent A set R of literals is called **consistent** iff no variable and its negation are both in R. Otherwise, R is said to be **inconsistent**.

Example: $\{x_1, \overline{x}_2, x_4, \overline{x}_7, \overline{x}_9\}$ is a consistent set, but $\{x_1, \overline{x}_2, x_4, \overline{x}_7, x_7\}$ is an inconsistent set.

Induced Assignment If $R = \{l_1, \ldots, l_n\}$ is a consistent set of literals, then α_R is called the **(partial) assignment induced by R** and is defined by $\alpha(l_i) = 1$ for all $l_i \in R$.

Example: the assignment induced by $R = \{x_1, \overline{x}_2, x_4, \overline{x}_7, \overline{x}_9\}$ is

$$\alpha = (x_1 = 1, x_2 = 0, x_4 = 1, x_7 = 0, x_9 = 0).$$

5 The 2SAT decision problem

In this section we introduce the 2SAT decision problem and show it is polynomial-time Turing reducible to Reachability. Afterwards, we improve the algorithm so that it no longer makes explicit queries to Reachability.

Definition 5.1. A **binary disjunctive clause** is a Boolean formula of the form

$$l_1 \vee l_2,$$

where l_1 and l_2 are literals. The clause evaluates to 1 in case either l_1 or l_2 (or both) is assigned 1.

Definition 5.2. An instance of the 2SAT decision problem consists of a set $\mathcal{C}$ of binary disjunctive clauses. The problem is to decide if there is an assignment α over the variables in $\mathcal{C}$, such that every clause $(l_1 \vee l_2)$ in $\mathcal{C}$ evaluates to 1 under α. If such an assignment α exists, then it is said to be a **satisfying assignment** and we say $\mathcal{C}$ is **satisfiable**. Otherwise, $\mathcal{C}$ is said to be **unsatisfiable**. Finally, the 2SAT decision problem is the problem of deciding whether a set $\mathcal{C}$ of clauses is sastisfiable.

Simplified clause notation. In what follows, we often simplify the clause notation by writing each clause $(l_1 \vee l_2)$ as (l_1, l_2).

Example 5.3. Provide a satisfying assignment for

$$\mathcal{C} = \{(x_2, \overline{x}_3), (x_1, \overline{x}_2), (x_3, x_4), (\overline{x}_2, \overline{x}_3), (\overline{x}_1, \overline{x}_4)\}.$$

We now demonstrate how to Turing reduce `2SAT` to `Reachability`, meaning that we can solve an instance of `2SAT` by making queries to a `Reachability`-oracle.

Definition 5.4. Let $\mathcal{C}$ be an instance of `2SAT`, and defined over the variables $x_1, x_2, \ldots, x_n$. Then the **implication graph** of $\mathcal{C}$ is defined as the directed graph $G_{\mathcal{C}} = (V, E)$, where

$$V = \{x_1, x_2, \ldots, x_n, \overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n\}$$

and each clause $(l_1 \vee l_2)$ produces the two directed edges $(\overline{l_1}, l_2), (\overline{l_2}, l_1) \in E$.

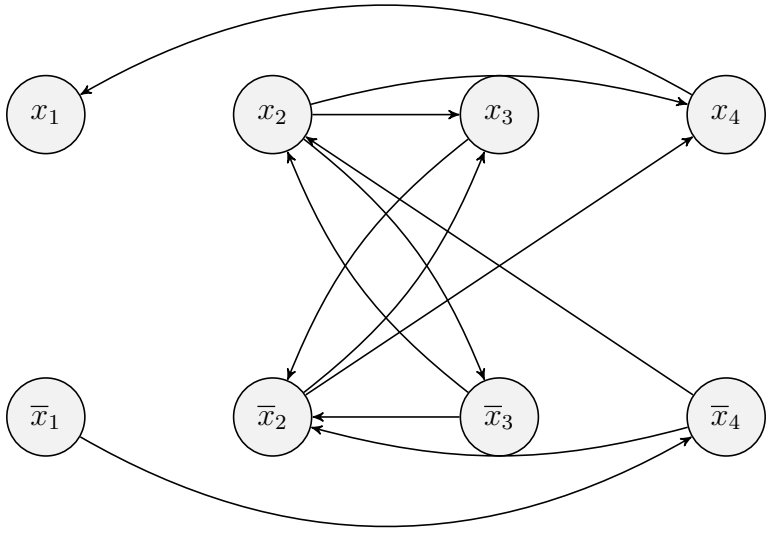
The idea behind the two edges formed from clause $c = (l_1 \vee l_2)$ is that c is logically equivalent to both the implication $\overline{l_1} \to l_2$ and its contrapositive $\overline{l_2} \to l_1$. Thus, for each edge (l_1, l_2) of $G_{\mathcal{C}}$, when l_1 is assumed true, then l_2 must also be true. This is so because (l_1, l_2) corresponds with the clause $(\bar{l}_1 \vee l_2)$ and the truth of l_1 forces the truth of l_2, since, according to the clause, either l_1 must be false or l_2 must be true.

Example 5.5. Verify that $l_1 \vee l_2$ is logically equivalent to both $\overline{l_1} \to l_2$ and $\overline{l_2} \to l_1$.

Solution.

Example 5.6. Draw the implication graph for the set $\mathcal{C}$ of clauses listed in the following table.

Clause	**Implication**	**Contrapositive**
$(\overline{x}_2, x_4)$	$x_2 \to x_4$	$\overline{x}_4 \to \overline{x}_2$
$(\overline{x}_2, \overline{x}_3)$	$x_2 \to \overline{x}_3$	$x_3 \to \overline{x}_2$
$(\overline{x}_2, x_3)$	$x_2 \to x_3$	$\overline{x}_3 \to \overline{x}_2$
(x_2, x_3)	$\overline{x}_2 \to x_3$	$\overline{x}_3 \to x_2$
(x_2, x_4)	$\overline{x}_2 \to x_4$	$\overline{x}_4 \to x_2$
$(x_1, \overline{x}_4)$	$\overline{x}_1 \to \overline{x}_4$	$x_4 \to x_1$



Implication Graph $G_{\mathcal{C}}$

Given the correspondence of a 2SAT instance $\mathcal{C}$ with an implication graph $G_{\mathcal{C}}$ whose number of vertices is twice the number n of variables, and whose number of edges is twice the number m of clauses, it seems appropriate to let $m = |\mathcal{C}|$ and n represent the size parameters for 2SAT.

Proposition 5.7. Given implication graph $G_{\mathcal{C}}$ and path

$$P = l_1, \ldots, l_n,$$

in $G_{\mathcal{C}}$, there is another path, called the **contrapositive** of P:

$$\overline{P} = \bar{l}_n, \ldots, \bar{l}_1.$$

Proof. If (x, y) is an edge of $G_{\mathcal{C}}$, then so is its contrapositive $(\overline{y}, \overline{x})$. Thus, every edge in a path from l_1 to l_n corresponds with its contrapositive in the path from $\bar{l}_n$ to $\bar{l}_1$. □

Theorem 5.8. 2SAT instance $\mathcal{C}$ is satisfiable iff there does not exist an **inconsistent cycle** in $G_{\mathcal{C}}$, i.e. a cycle that contains a variable and its negation.

Before proving Theorem 5.8, we use it to provide an algorithm showing that 2SAT can be polynomial-time Turing reduced to Reachability by making at most $2n$ queries. The algorithm uses the observation that an inconsistent cycle that includes x and $\overline{x}$ is equivalent to two paths: one from x to $\overline{x}$, and another from $\overline{x}$ to x. Thus, checking if $G_{\mathcal{C}}$ has an inconsistent cycle that includes x and $\overline{x}$ can be done with two queries to a Reachability-oracle.

2SAT Algorithm

Input: 2SAT instance $\mathcal{C}$.

Ouput: true iff $\mathcal{C}$ is satisfiable.

Construct $G_{\mathcal{C}}$.

For each $x \in \text{var}(\mathcal{C})$,

If reachable$(G_{\mathcal{C}}, x, \overline{x})$ and reachable$(G_{\mathcal{C}}, \overline{x}, x)$, then return false.

Return true.

Example 5.9. Consider a 2SAT instance $\mathcal{C}$ for which $G_{\mathcal{C}}$ has the cycle

$$C = \overline{x}_1, x_3, \overline{x}_5, x_1, x_2, \overline{x}_1.$$

Verify that $\mathcal{C}$ is unsatisfiable.

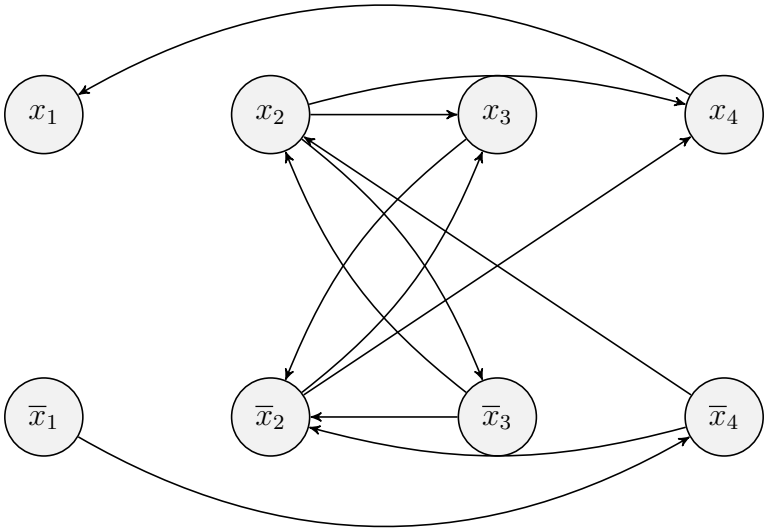
The following proposition provides the key to finding a satisfying assignment for 2SAT instance $\mathcal{C}$ in case all of $G_{\mathcal{C}}$'s cycles are consistent. It relies on the notion of a *reachability set* for a graph vertex.

Definition 5.10. Let $G = (V, E)$ be a graph and $v \in V$ a vertex of G. Then the **reachability set** of v in G is the set of all vertices that can be reached by v along some path (regardless of its length).

Example 5.11. Verify that the reachability set for vertex x_2 of the implication graph in Example 5.6 is equal to

$$R = \{x_2, x_3, \overline{x}_3, x_4, \overline{x}_2, x_1\}.$$

Is R a consistent or inconsistent set of literals?



Proposition 5.12. Given 2SAT instance $\mathcal{C}$, implication graph $G_{\mathcal{C}}$, and vertex/literal l, if R_l, the reachability set of l, does not contain $\bar{l}$, then the following statements are true.

1. R_l is a consistent set of literals.
2. Assignment α_{R_l} satisfies every clause in $\mathcal{C}$ that depends on at least one variable assigned by α_{R_l}.

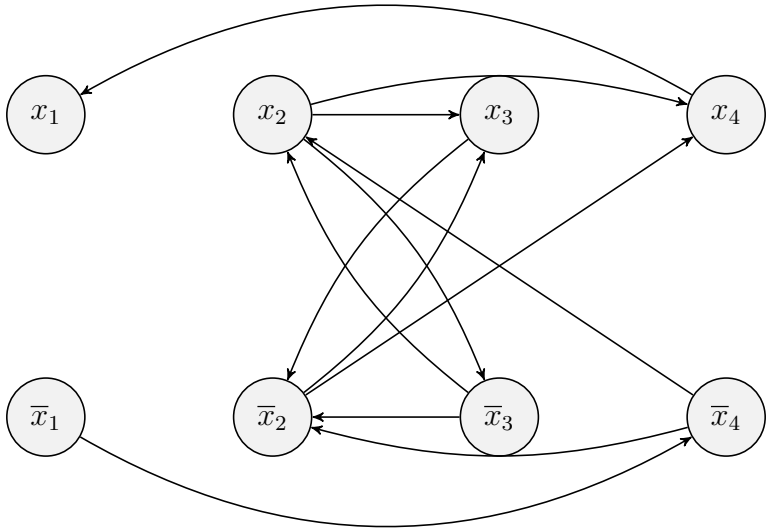
Proof of Statement 1: By way of contradiction, assume there is a variable x for which $x \in R_l$ and $\overline{x} \in R_l$. Then there is a path P_1 from l to x and a path P_2 from l to $\overline{x}$. Hence, $P_1 \cdot \overline{P}_2$ (i.e. the concatenation of P_1 with the contrapositive of P_2) yields a path from x to $\overline{x}$, a contradiction.

Proof of Statement 2: Consider a clause $c = (l_1, l_2)$ of $\mathcal{C}$ for which either i) $l_1 \in R_l$, ii) $l_2 \in R_l$, iii) $\bar{l}_1 \in R_l$, or iv) $\bar{l}_2 \in R_l$. If either i) or ii) is true, then c is satisfied by α_{R_l}, since any literal in R is assigned 1 by α_R. Now suppose iii) is true (case iv is identical). By the definition of $G_{\mathcal{C}}$, clause c contributes the edge

$$(\bar{l}_1, l_2).$$

Thus, since $\bar{l}_1$ is reachable from l, so is l_2, and c is satisfied by α_{R_l} since l_2 is assigned 1 by α_{R_1}. Hence, α_{R_l} satisfies any clause that contains a literal l' for which either $l' \in R_l$ or $\overline{l'} \in R_l$. □

Example 5.13. For the implication graph below, verify that the reachability set for vertex $\overline{x}_1$ contains both x_2 and $\overline{x}_2$, and so it must also contain x_1.

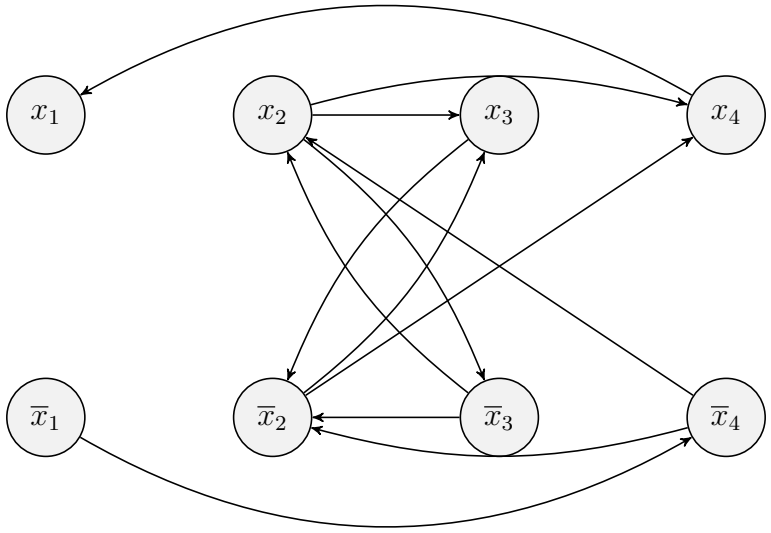


Example 5.14. Verify that the reachability set for vertex x_4 of the implication graph in Example 5.6 is equal to the consistent set

$$R = \{x_4, x_1\},$$

and show that α_R satisfies all clauses of $\mathcal{C}$ that use one of the variables from R.

Clause	**Implication**	**Contrapositive**
$(\overline{x}_2, x_4)$	$x_2 \to x_4$	$\overline{x}_4 \to \overline{x}_2$
$(\overline{x}_2, \overline{x}_3)$	$x_2 \to \overline{x}_3$	$x_3 \to \overline{x}_2$
$(\overline{x}_2, x_3)$	$x_2 \to x_3$	$\overline{x}_3 \to \overline{x}_2$
(x_2, x_3)	$\overline{x}_2 \to x_3$	$\overline{x}_3 \to x_2$
(x_2, x_4)	$\overline{x}_2 \to x_4$	$\overline{x}_4 \to x_2$
$(x_1, \overline{x}_4)$	$\overline{x}_1 \to \overline{x}_4$	$x_4 \to x_1$



Proof of Theorem 5.8. The statement of the theorem is of the form $P \leftrightarrow Q$, where P stands for "$\mathcal{C}$ is satisfiable", and Q stands "$G_{\mathcal{C}}$ has only consistent cycles". We can thus break it down to two different statements: $P \rightarrow Q$ and $Q \rightarrow P$.

We first prove $P \rightarrow Q$ via an indirect proof. **Assume** $\neg Q$: "$G_{\mathcal{C}}$ has an inconsistent cycle". **Show** $\neg P$: $\mathcal{C}$ is unsatisfiable. To this end, assume that there exists a variable x such that $\overline{x}$ is reachable from x and x is reachable from $\overline{x}$ in the implication graph $G_{\mathcal{C}}$. Then there are edge sequences in $G_{\mathcal{C}}$ of the form

$$(x, l_1), (l_1, l_2), \ldots, (l_{r-1}, l_r), (l_r, \overline{x})$$

and

$$(\overline{x}, \hat{l}_1), (\hat{l}_1, \hat{l}_2), \ldots, (\hat{l}_{s-1}, \hat{l}_s), (\hat{l}_s, x).$$

The first edge sequence implies that $\mathcal{C}$ is not satisfiable when x is assigned to 1. For example, $(x, l_1) \in E$ implies that either $(\overline{x}, l_1)$ or $(l_1, \overline{x})$ is a literal of $\mathcal{C}$. Then the assignment of 1 to x forces an assignment of 1 to l_1. Similar reasoning shows that this in turn forces an assignment of 1 to l_2, and, using this reasoning through the entire edge sequence, we see that l_r is forced to have an assignment of 1. But edge $(l_r, \overline{x})$ corresponds with the literal $(\overline{l_r}, \overline{x})$. And if l_r is forced to 1, then this literal cannot be satisifed, since x was already assigned 1. Hence, no satisfying assignment for $\mathcal{C}$ can assign x to 1. A similar argument using the second edge sequence shows that no satisfying assignment for $\mathcal{C}$ can assign x to 0. Therefore, $\mathcal{C}$ is unsatisfiable.

We now prove $Q \rightarrow P$ using a direct proof and mathematical induction. **Assume** Q: "$G_{\mathcal{C}}$ has only consistent cycles". **Show** P: $\mathcal{C}$ is satisfiable. We show G is satisfiable by induction on the number n of variables appearing in $\mathcal{C}$.

Basis Step. $\mathcal{C}$ has one variable x. Since there are no inconsistent cycles with both x and $\overline{x}$ it must be the case that $\mathcal{C}$ either consists of the single clause (x, x), or of the single clause $(\overline{x}, \overline{x})$. In either case $\mathcal{C}$ is satisfiable.

Induction Step. Assume that any `2SAT` instance having $n-1$ or fewer variables and only consistent cycles in its implication graph is satisfiable, for some $n \geq 2$. Let $\mathcal{C}$ be an instance with n variables and only consistent cycles. Then there must be a literal l for which there is no path from l to $\bar{l}$. By Proposition 5.12 R_l is consistent and α_{R_l} satisfies every clause that has a variable that gets assigned by α_{R_l}.

Now let C_{R_l} denote the set of clauses satisfied by α_{R_l}. Then consider the new `2SAT` instance $\mathcal{C}' = \mathcal{C} - C_{R_l}$ that is the result of removing the clauses in C_{R_l} from $\mathcal{C}$. Then $\mathcal{C}'$ has fewer than n variables, and since $G'_{\mathcal{C}}$ is a subgraph of $G_{\mathcal{C}}$, it follows that $G'_{\mathcal{C}}$ has only consistent cycles. Therefore, by the inductive assumption, $\mathcal{C}'$ is satisfiable. FInally, if α' is a satisfying assignment for $\mathcal{C}'$, then $\alpha' \cup \alpha_R$ satisfies $\mathcal{C}$. □

The proof of Theorem 5.8 suggests the following recursive algorithm for determining the satisfiability of `2SAT` instance $\mathcal{C}$. The algorithm returns a non-empty satisfying assignment if $\mathcal{C}$ is satisfiable, and returns $\emptyset$ otherwise.

Improved 2SAT Algorithm

Name: `sat2`

Input: `2SAT` instance $\mathcal{C}$ and a pointer α to an assignment (initially, $\emptyset$).

Output: `true` iff $\mathcal{C}$ is satisfiable.

Side Effect: if $\mathcal{C}$ is satisfiable, then α points to a sat assignment. Otherwise, $\alpha \leftarrow \emptyset$.

//Base Case:

If $\mathcal{C} = \emptyset$, return `true`. //an empty set of clauses is considered satisfied

//Recursive case:

Construct $G_{\mathcal{C}}$.

Choose a literal l and compute R, the set of literals reachable from l.

If R is consistent,

> Update α: $\alpha \leftarrow \alpha \cup \alpha_R$.
> $\mathcal{C}' \leftarrow \mathcal{C} - C_R$, where C_R denotes all clauses satisfied by α_R.
> Return `sat2`$(\mathcal{C}', \alpha)$.

Else

> Compute $\overline{R}$, the set of literals reachable from $\bar{l}$.
> If $\overline{R}$ is consistent,
>
> > Update α: $\alpha \leftarrow \alpha \cup \alpha_{\overline{R}}$.
> > $\mathcal{C}' \leftarrow \mathcal{C} - C_{\overline{R}}$, where $C_{\overline{R}}$ denotes all clauses satisfied by $\alpha_{\overline{R}}$.
> > Return `sat2`$(\mathcal{C}', \alpha)$.
>
> Else
>
> > $\alpha \leftarrow \emptyset$.
> > Return `false`.

By modifying the algorithm so that the steps needed to compute each reachability set (for l and $\bar{l}$) are alternated, we can guarantee that the total work performed is at most $O(m + n)$ steps.

Example 5.15. Use the improved 2SAT algorithm to determine a satisfying assignment for

$$\mathcal{C} = \{(x_2, \overline{x}_3), (x_1, \overline{x}_2), (x_3, x_4), (\overline{x}_2, \overline{x}_3), (\overline{x}_1, \overline{x}_4), (x_5, x_6), (\overline{x}_5, \overline{x}_6), (\overline{x}_1, x_6)\}.$$

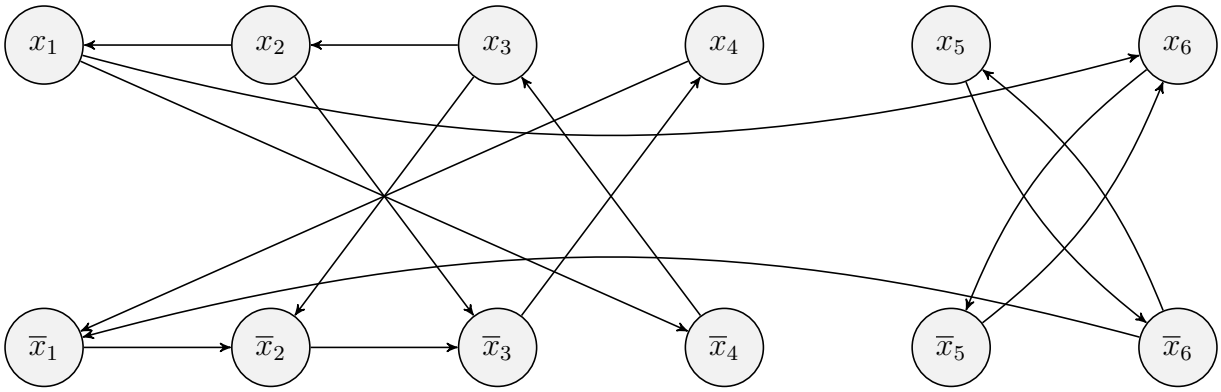
Start off by choosing $l = x_1$.

Solution.

1. Compute the edges for $G_{\mathcal{C}}$.

Clause	**Edges**
$(x_2, \overline{x}_3)$	$\overline{x}_2 \to \overline{x}_3,\ x_3 \to x_2$
$(x_1, \overline{x}_2)$	$\overline{x}_1 \to \overline{x}_2,\ x_2 \to x_1$
(x_3, x_4)	$\overline{x}_3 \to x_4,\ \overline{x}_4 \to x_3$
$(\overline{x}_2, \overline{x}_3)$	$x_2 \to \overline{x}_3,\ x_3 \to \overline{x}_2$
$(\overline{x}_1, \overline{x}_4)$	$x_1 \to \overline{x}_4,\ x_4 \to \overline{x}_1$
(x_5, x_6)	$\overline{x}_5 \to x_6,\ \overline{x}_6 \to x_5$
$(\overline{x}_5, \overline{x}_6)$	$x_5 \to \overline{x}_6,\ x_6 \to \overline{x}_5$
$(\overline{x}_1, x_6)$	$x_1 \to x_6,\ \overline{x}_6 \to \overline{x}_1$

2. Draw the implication graph



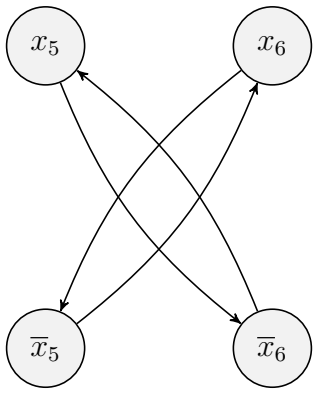
3. Compute $R_{x_1} = \{x_1, \overline{x}_1, x_2, \overline{x}_2, x_3, \overline{x}_3, x_4, \overline{x}_4, \overline{x}_5, x_6\}$ which is inconsistent.

4. Compute $R_{\overline{x}_1} = \{\overline{x}_1, \overline{x}_2, \overline{x}_3, x_4\}$ which is consistent. Verify that

$$\alpha_{R_{\overline{x}_1}} = (x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1)$$

satisfies all clauses that involve variables x_1, x_2, x_3, x_4.

5. Draw the reduced implication graph $G_{\mathcal{C}'}$ where $\mathcal{C}' = \{(x_5, x_6), (\overline{x}_5, \overline{x}_6)\}$.



6. Compute $R_{x_5} = \{x_5, \overline{x}_6\}$ which is consistent. Verify that $\alpha_{R_{x_5}} = (x_5 = 1, x_6 = 0)$ satisfies both clauses in $\mathcal{C}'$.

7. Final satisfying assignment:

$$\alpha = \alpha_{R_{\overline{x}_1}} \cup \alpha_{R_{x_5}} = (x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 1, x_6 = 0).$$

6 Finding Maximum Network Flows

Network Flow

Let $G = (V, E, c, s, t)$ be a directed network, where $c : E \to R^+$ determines the **capacity** of each edge, $s \in V$ is the designated **source vertex** and $t \in V$ is the designated **destination vertex**. Throughout this section we use the convention of not drawing a graph edge in case it's capacity equals zero. A **flow** through the network is a function $f : E \to R^+$ with the following properties:

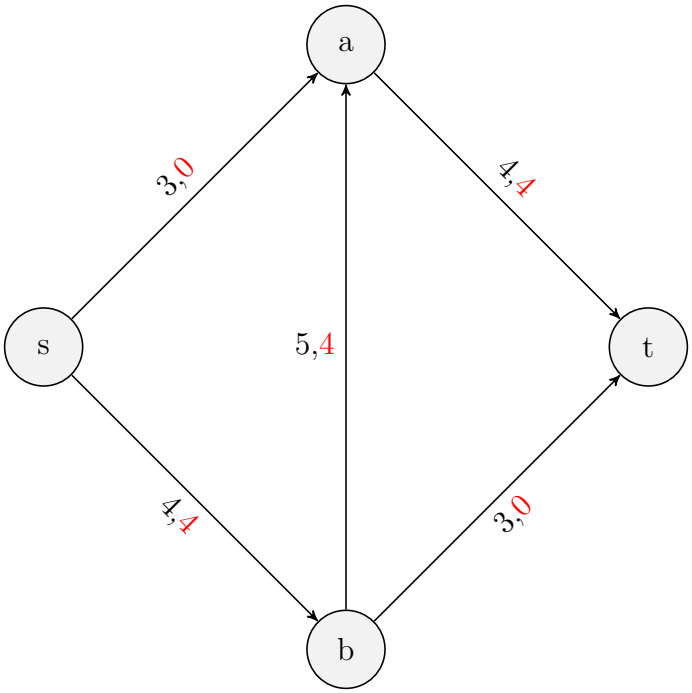
1. For every $e \in E$, $f(e) \leq c(e)$. In other words, the flow through an edge should not exceed the edge's capacity.

2. For every vertex v, let $E^+(v)$ equal the set of edges that end at v, and $E^-(v)$ the set of edges that start at v. Then for every intermediate vertex $v \in V - \{s, t\}$, we have

$$\sum_{e \in E^+(v)} f(e) = \sum_{e \in E^-(v)} f(e).$$

   In other words, the total flow entering an intermediate vertex v must equal the total flow leaving v. Any vertex that has the above property is said to be **flow conserving**.

Example 6.1. For the directed network below, the left edge label is the edge capacity, while the right edge-label is the flow value for a flow f.



The **size** of a flow f through a network, denoted $s(f)$, is defined as

$$s(f) = \sum_{e \in E^-(s)} f(e),$$

and represents the total flow that is leaving source s. For the flow f in Example 6.1 we have $s(f) = 0 + 4 = 4$.

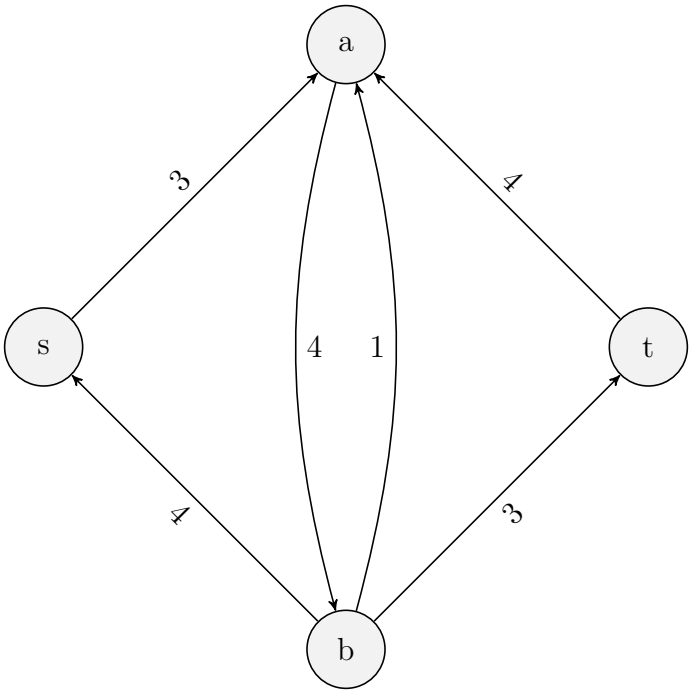
An instance of the `Max Flow` optimization problem is a directed network $G = (V, E, c, s, t)$ and the problem is to find the size of the **maximum flow** f, i.e., the size of a flow f for which $s(f) \geq s(f')$ for all other flows f'.

We now describe the **Ford-Fulkerson algorithm** that is used for finding a maximum flow for a network. The algorithm is quite relevant to the topic of Turing reducibility since, like the `2SAT` algorithm, it works by making a sequence of queries to a `Reachability`-oracle. However, each query answer will provide an actual path in case the query answer is `yes`. It's this path that will be used to improve the existing network flow.

Given network $G = (V, E, c, s, t)$ and flow f through G, the key idea behind the algorithm is to define the **residual network** G_f. with respect to G and f, where $G_f = (V, E', c', s, t)$ is defined once E' and c' are defined as follows. Let $e = (u, v) \in E'$ be given. Then one of the following must be true.

1. $e \in E$ and $f(e) < c(e)$. In this case we refer to e as a **forward edge** since it belongs to the original network. Moreover, its capacity $c'(e) = c(e) - f(e)$ equals the remaining unused capacity of e.
2. $e^r = (v, u) \in E$ and $f(e^r) > 0$. In this case we refer to e as a **backward edge** because it is oriented in the opposite direction as $e^r \in E$. Moreover, its capacity $c'(e) = f(e^r)$ equals the amount of flow passing through e^r and thus is the amount of flow that can be redirected away from e^r.

Example 6.2. The network below is the residual network G_f for the network G and flow f shown in Example 6.1.



The following theorem establishes how the residual network G_f is used to i) determine whether f is a maximum flow for G and ii) obtain a larger flow in case f is not a maximum flow.

Theorem 6.3. Given network $G = (V, E, c, s, t)$ and flow f through G, f is a maximum flow iff t is not reachable from s in G_f. In case t is reachable from s via some **augmenting path** P, and letting κ equal the minimum capacity of any edge traversed by P, then a new flow $\Delta(f, P)$ may be defined through G as follows. For each $e \in E$,

$$\Delta(f,P)(e) = \begin{cases} f(e) + \kappa & \text{if } e \text{ traversed by } P \\ f(e) - \kappa & \text{if } e^r \text{ traversed by } P \\ f(e) & \text{otherwise} \end{cases}$$

Note: recall that e^r refers to the backward edge associated with network edge $e \in E$. In other words, if $e = (u, v)$, then $e^r = (v, u)$.

Before proving Theorem 6.3, we summarize how to obtain the new flow $\Delta(f, P)$ from the existing flow f and augmenting path P.

1. If $e \in E$ is a forward edge traversed by P, then add κ units of flow to e.
2. If $e \in E$ and backward edge e^r is traversed by P, then subtract κ units of flow from e.
3. if neither $e \in E$ nor its backward version e^r is traversed by P, then do not change the flow through e.

Proof. The statement being asserted by Theorem 6.3 has the form $Q \leftrightarrow R$, where Q is the statement "f is a maximum flow", and R is the statement "t is not reachable from s in G_f".

We first prove $Q \rightarrow R$ using an indirect proof. Assume $\overline{R}$: t is reachable from s via some path P and let κ denote the minimum capacity of any edge traversed by P. Prove $\overline{Q}$: $f' = \Delta(f, P)$, as defined in the theorem, is a flow for G that exceeds f. To this end we first must show that, for all $e \in E$,

$$0 \leq f'(e) \leq c(e).$$

Case 1: e is traversed in P. Then $0 \leq f'(e) = f(e) + \kappa \leq f(e) + (c(e) - f(e)) = c(e)$.

Case 2: e^r is traversed in P. Then $c(e) \geq f'(e) = f(e) - \kappa \geq f(e) - f(e)) \geq 0$.

Case 3: neither e nor e^r is traversed by P. Then $0 \leq f'(e) \leq c(e)$ since $f'(e) = f(e)$, and we are assuming that $f(e)$ satisfies these inequalities since it is a known flow.

Lastly, we must show that all vertices in $V - \{s, t\}$ remain flow-conserving with respect to f'. The only vertices for which the total flow entering and leaving a vertex may have changed are those internal vertices that are visited by P. Let v be such a vertex and let e_{in} (respectively, e_{out}) be the edge traversed by P that enters (respectively, leaves) v. Then there are four cases to consider depending on orientation (forward or backwards) of both edges.

Case 1: e_{in} and e_{out} are both forward edges. Then $e_{\text{in}}, e_{\text{out}} \in E$ which means P sends an additional κ units of flow both into and out of v, and so flow is conserved.

Case 2: e_{in} and e_{out} are both backward edges. Then $e^r_{\text{in}}, e^r_{\text{out}} \in E$ which means P removes κ units of flow from e_{in} that was leaving v and κ units of flow from e_{out} that was entering v, and so flow is conserved.

Case 3: e_{in} is a forward edge and e_{out} is a backward edge. Then $e_{\text{in}}, e^r_{\text{out}} \in E$ which means P sends an additional κ units of flow into v via e_{in} and removes from e^r_{out} κ units of flow that was entering v, and so flow is conserved.

Case 4: e_{in} is a backward edge and e_{out} is a forward edge. Then $e^r_{\text{in}}, e_{\text{out}} \in E$ which means P removes κ units of flow from e^r_{in} that was leaving v and adds κ units of flow to e_{out} that is leaving v. In other words, κ units of flow from v have been re-directed from e^r_{in} to e_{out}, and so flow is conserved.

We now turn our attention to proving $R \rightarrow Q$ using an indirect proof. Assume $\overline{Q}$: f is not a maximum flow for G. Prove $\overline{R}$: t is reachable from s. Since f is not maximum, there exists a flow g for G with $s(g) > s(f)$. We use g to define a positive flow δ over the residual network G_f. Let $e \in E$ be an edge of both G and G_f. Thus, e is a forward edge in G_f and we let e^r denote its associated backward edge. We now define δ as follows.

1. if $f(e) = g(e)$, then $\delta(e) = \delta(e^r) = 0$,
2. if $f(e) > g(e)$, then $\delta(e) = 0$, while $\delta(e^r) = f(e) - g(e)$, and
3. if $f(e) < g(e)$, then $\delta(e^r) = 0$, while $\delta(e) = g(e) - f(e)$.

We leave it as an exercise to check that δ obeys the edge-capacity limits defined by G_f. Moreover, let $v \in V - \{s, t\}$ be an intermediate vertex of G_f. We also leave it as an exercise to check that

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

Finally, since $s(g) > s(f)$, it follows that $s(\delta) > 0$ over G_f. Thus δ is a positve flow over G_f, and, by Exercise 19, there is a path from s to t in G_f, i.e., t is reachable from s. □

We now summarize the Ford-Fulkerson algorithm for finding a maximum flow for a network.

Ford-Fulkerson Algorithm

Input network $G = (V, E, f, s, t)$.

Initialize flow f, where $f(e) = 0$, for all $e \in E$.

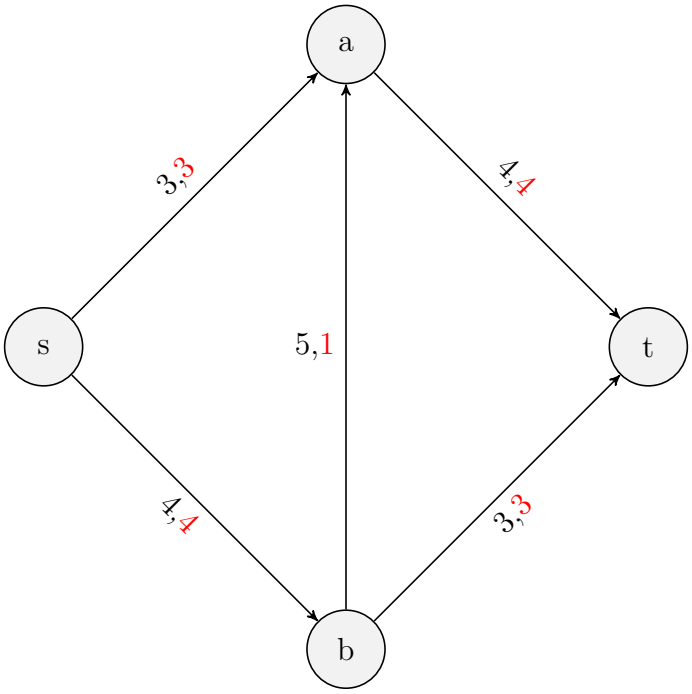
While `reachable`(G_f, s, t) is true,

Let P be a path from s to t in G_f.
Update f: $f \leftarrow \Delta(f, P)$.

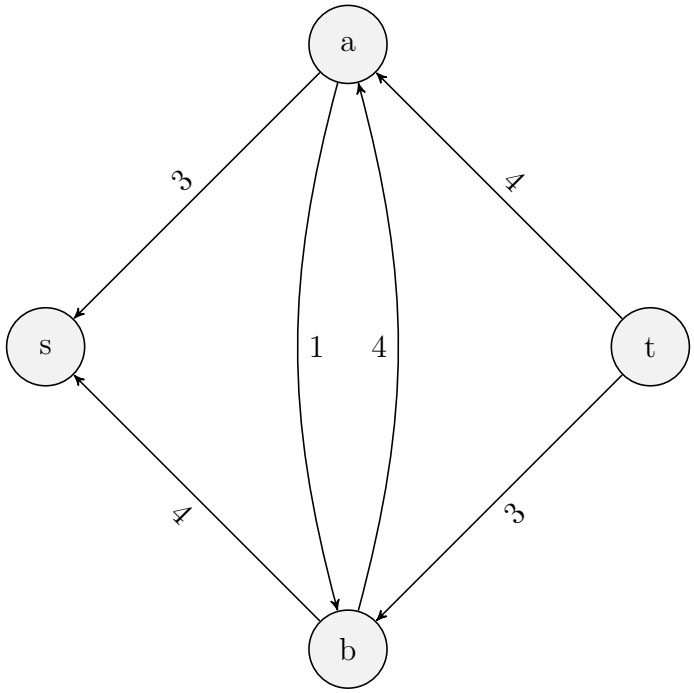
Return f.

Example 6.4. Starting with the flow f provided in Example 6.1, use the Ford-Fulkerson algorithm to find a maximum flow for G provided in the example.

Solution. Based on the resisual graph G_f shown in Example 6.2. We see that t is reachable from s via path $P = s, a, b, t$, and for which $\kappa = \min(3, 4, 3) = 3$. The following graph now shows G with flow $f_2 = \Delta(f, P)$. The changes made from f to f_2 can be described by following P: add 3 units to (s, a), remove 3 units from (b, a), and add 3 units to (b, t).



We now provide G_{f_2} which shows that t is not reachable from s. Therefore, f_2 is a maximum flow for G.



For the running time of this algorithm, one can show that, if each augmenting path is obtained via a breadth-first traversal of G_f starting at s, then the maximum flow can be obtained in $O(nm^2)$ steps. Also, there do exist better algorithms for solving `Max Flow`, including one that requires $O(n^3)$ steps.

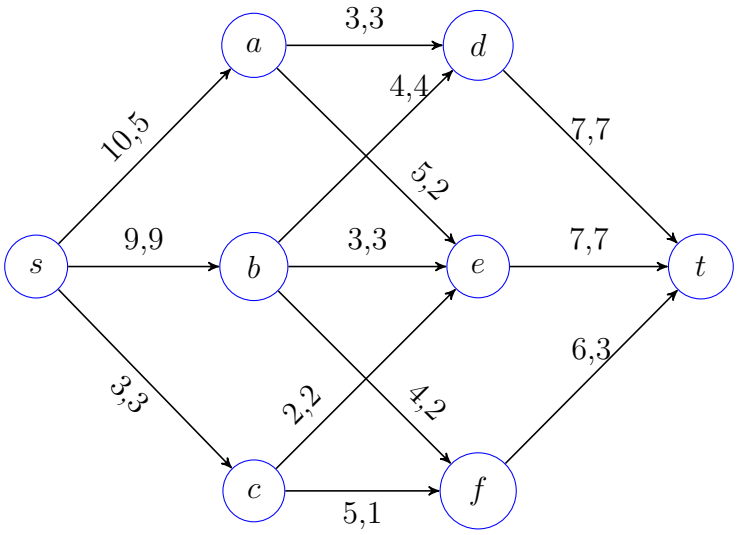


Figure 1: The network for Example 6.5.

Example 6.5. Consider the network $G = (V, E, c, s, t)$ shown in Figure 1, along with a flow f, in which each edge e is labeled with two numbers: the edge capacity $c(e)$, and the flow value $f(e)$. Draw the residual network G_f and use it to determine an augmenting path P from s to t, and label G with the new flow $f' = \Delta(f, P)$, by crossing out any no-longer-valid flow labels and replacing them with new ones.

Exercises

1. Does Marcia's algorithm from Example 2.1 prove that `Multiply` is polynomial-time Turing reducible to `Add`? Explain.

2. Prove that if there is an algorithm that solves problem A in polynomial-time, then $A \leq_{\mathrm{T}}^{\mathrm{p}} B$ for any problem B.

3. Consider the following functions.

```
//Turing reduces A to B
Boolean solve_A_with_B(int n)
{
    //Solve instance n of A by making B-queries
    return query_B(n*n) || query_B(n+6);
}

//Turing reduces B to C
Boolean solve_B_with_C(int n)
{
    //Solve instance n of B by making C-queries
    return !query_C(n+8) && query_C(5*n);
}
```

   Implement a third function `solve_A_with_C` that is a witness to $A \leq_{\mathrm{T}} C$. Note: your function must take an instance n of A and return a Boolean decision that uses logic and is allowed to only make C-queries.

4. Use the previous exercise as inspiration for proving the general result that $\leq_{\mathrm{T}}$ reducibility relation is transitive. In other words, if $A \leq_{\mathrm{T}} B$ and $B \leq_{\mathrm{T}} C$, then $A \leq_{\mathrm{T}} C$.

5. Repeat the previous exercise, but replace $\leq_{\mathrm{T}}$ with $\leq_{\mathrm{T}}^{\mathrm{p}}$

6. For the `Reachability` algorithm, use math induction to prove that any vertex x that gets marked is reachable from u. Hint: assign an index to each marked vertex that represents the distance of that vertex from u. In particular, assign u index 0. Then if vertex w has assigned index i and (w, x) is the edge responsible for the marking of x, then assign x index $i + 1$. Perform the induction on the index i assigned to vertex x.

7. For the directed graph $G = (V, E)$, where

$$V = \{a, b, c, d, e, f, g, h, i, j, k\}$$

   and the edges are given by

$$E = \{(a, b), (a, c), (b, c), (b, d), (b, e), (b, g), (c, g), (c, f),$$
$$(d, f), (f, g), (f, h), (g, h), (i, j), (i, k), (j, k)\},$$

   use the Reachability Algorithm to determine if vertex k is reachable from vertex a. Show the contents of the FIFO queue Q at each stage of the algorithm.

8. Find a satisfying assignment for the set of clauses

$$\mathcal{C} = \{(x_1, x_2), (\overline{x}_3, \overline{x}_4), (x_3, \overline{x}_5), (x_2, x_5), (\overline{x}_2, x_3), (\overline{x}_1, \overline{x}_4), (\overline{x}_1, \overline{x}_5), (\overline{x}_2, x_5)\}.$$

9. For `2SAT` instance $\mathcal{C}$, suppose you make the query `reachable`$(G_{\mathcal{C}}, x_3, \overline{x}_3)$ to a `Reachability` oracle who answers the query with "yes". Assuming $\mathcal{C}$ is satisfiable, what can you say about a satisfying assignment for $\mathcal{C}$? Explain.

10. For some `2SAT` instance $\mathcal{C}$, is it possible to know with certainty whether or not $\mathcal{C}$ is satisfiable by making exactly one query to a `Reachability` oracle and assuming no other knowledge about $\mathcal{C}$, including its size? Defend your answer.

11. Draw the implication graph for the following set of CNF clauses.

$$(\overline{x}_2, \overline{x}_3), (x_2, \overline{x}_4), (x_1, \overline{x}_3), (x_2, x_3), (x_1, x_4), (\overline{x}_1, x_4), (x_1, \overline{x}_2).$$

Perform the Improved 2SAT Algorithm to determine a satisfying assignment for this set of clauses. For the line that asks you to "choose a literal l", choose $l = x_1$.

12. Repeat the previous problem, but now add the additional clause $(\overline{x}_2, x_3)$. Verify that there is now a cycle in the implication graph which contains a variable and its negation. Which variable is it?

13. Draw the implication graph for the following set of CNF clauses.

$$\mathcal{C} = \{(x_2, \overline{x}_4), (\overline{x}_2, x_5), (x_4, x_6), (\overline{x}_2, \overline{x}_4), (\overline{x}_5, \overline{x}_6), (\overline{x}_1, x_3), (x_1, \overline{x}_3), (x_3, \overline{x}_5)\}.$$

Perform the Improved 2SAT Algorithm to determine a satisfying assignment for this set of clauses. For the line that asks you to "choose a literal l", choose the positive literal of least index. For example, at the top level of recursion that would be $l = x_1$ since x_1 positive and has smallest index 1. Hint: the recursive case should be executed twice.

14. In the `2SAT` algorithm, suppose the oracle answers `yes` to `reachable`$(G_{\mathcal{C}}, \overline{x}_3, x_3)$, but `no` to `reachable`$(G_{\mathcal{C}}, x_3, \overline{x}_3)$. Then if $\mathcal{C}$ is a unique satisfying assignment α, then what can you say about α?

15. A network consists of the following directed and weighted edges:

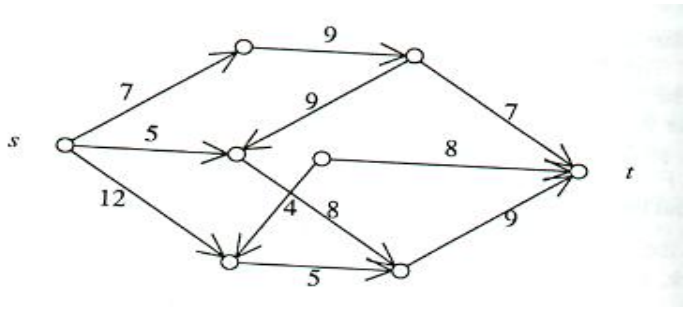
$$(s, a, 10), (s, b, 10), (a, c, 10), (b, d, 9), (b, e, 6), (c, b, 5), (c, t, 7), (d, e, 7), (d, t, 5), (e, t, 8).$$

Demonstrate the Ford-Fulkerson algorithm on this network with source vertex s, and destination vertex t. Assume an initial flow of

$$f_1 = (s, a, 5), (a, c, 5), (c, b, 5), (b, d, 5), (d, e, 5), (e, t, 5).$$

To make it interesting, for each round of the algorithm, choose an augmenting path P from s to t that has maximum *length*, i.e., number of edges traversed by P. Draw the sequence of residual networks and redraw the network/flow with each corresponding updated flow.

16. Repeat the previous exercise for the network shown below. When there is more than one augmenting path P from s to t, choose the one having maximum *length*. Assume an initial flow that is zero on all edges.



17. Given a network $G = (V, E, c, s, t)$ and a positive flow f through G, consider a the subgraph H induced by a set of flow-conserving vertices $I \subseteq V - \{s, t\}$. In other words, the vertex set of H is equal to I, while $e = (u, v)$ is an edge of H iff $e \in E$, and $u, v \in I$. Let $E^+(H)$ denote all edges $e = (u, v)$, such that $e \in E$, $u \notin I$, and $v \in I$, while $E^-(H)$ denotes all edges $e = (u, v)$, such that $e \in E$, $u \in I$, and $v \notin I$. Prove that

$$\sum_{e \in E^+(H)} f(e) = \sum_{e \in E^-(H)} f(e).$$

In other words, flow is conserved within a subgraph induced by flow-conserving vertices. Hint: use mathematical induction on the number of vertices in H.

18. Consider a weighted directed graph $G = (V, E, f)$, where $V = \{v_0, v_1, \ldots, v_{n-1}\}$, and $f : E \to R^+$ represents a positive flow through G. Moreover, assume vertex v_0 has zero in-degree and out-degree equal to one via the edge $e = (v_0, v_1)$ for which $f(e) > 0$. Hence, v_0 is not flow conserving since there is zero flow entering v_0, but $f(e) > 0$ flow leaving it. Prove that there must exist at least one other vertex in G that is non-flow-conserving. Hint: use a proof by contradiction together with the previous exercise.

19. Prove that if network $G = (V, E, c, s, t)$ admits a postive flow f, then there must exist a path P from s to t in G for which $f(e) > 0$ for every edge traversed by P. Hint: use the previous exercise.

20. In the $(R \to Q)$-part of the proof of Theorem 6.3 a function δ was defined on the edges of the residual network G_f. Prove that, for each $e \in E$, $\delta(e) \leq c(e) - f(e)$ and $\delta(e^r) \leq f(e)$. Conclude that δ obeys the first of the two properties needed to be classified as a flow through G_f.

21. In the $(Q \to R)$-part of the proof of Theorem 6.3 a function δ was defined on the edges of the residual network G_f. Prove that, for each vertex $v \in V - \{s, t\}$ the sum of all δ-values of edges entering v equals the sum of all δ-values of edges leaving v. Together with the previous exercise, conclude that δ is in fact a positive flow through G_f.

Exercise Solutions

1. Marcia's algorithm does *not* run in polynomial time. To see this, suppose that Marcia wants the answer to $m \times m$. Then the the size of this problem instance is $2\lfloor \log m \rfloor$. However, Marcia's algorithm will require the inclusion of $m - 1$ queries

$$m + m, 2m + m, \ldots, (m-1)m + m,$$

and $m - 1$ is exponential with respect to $\log m$. Thus, the number of algorithm steps grows exponentially with respect to the problem size.

2. Suppose A can be solved by some algorithm that runs in polynomial time. Let B be any other problem. Then $A \leq_{\mathrm{T}}^{\mathrm{p}} B$ is true in a trivial sense, since the algorithm that solves A makes zero queries to a B-oracle and so $A \leq_{\mathrm{T}}^{\mathrm{p}} B$ by definition of Turing reducible, since the definition states that there exists a polynomial-time algorithm for solving A that makes "**zero** or more queries" to a B-oracle.

3. We have the following function that proves $A \leq_{\mathrm{T}} C$.

```
//Turing reduces A to C
Boolean solve_A_with_C(int n)
{
    //Solve instance n of A by making C-queries
    return (!query_C((n*n)+8) && query_C(5*(n*n))) ||
         (!query_C((n+6)+8) && query_C(5*(n+6)));
}
```

4. Suppose $A \leq_{\mathrm{T}} B$. Then there is an algorithm $\mathcal{A}_{AB}$ that solves an instance of A by making queries to a B-oracel. Moreover, since $B \leq_{\mathrm{T}} C$, there is also an algorithm $\mathcal{A}_{BC}$ that solves an instance of B by making queries to a C-oracle.

   We now describe an algorithm $\mathcal{A}_{AC}$ that solves instances of A by making queries to a C-oracle. This algorithm is obtained by modifying $\mathcal{A}_{AB}$ as follows. For each B-query step query(y), where y is an instance of B, we replace this B-query step with a function call to $\mathcal{A}_{BC}(y)$, which is the answer returned by $\mathcal{A}_{BC}$ on input y. We may think of $\mathcal{A}_{BC}(y)$ as a function call that is being made within the body of $\mathcal{A}_{AB}$. Of course, the $\mathcal{A}_{BC}$ function has its own body of source code, which is now part of the $\mathcal{A}_{AC}$ code base. After modifying $\mathcal{A}_{AB}$ in this manner, notice that the only query steps in $\mathcal{A}_{AC}$ are found in the inserted $\mathcal{A}_{BC}$ code, and are of the form query(z), where z is a problem instance of C. In other words, all queries are to a C-oracle. Hence, $\mathcal{A}_{AC}$ is an algorithm that Turing reduces A to C.

5. From the solution to the previous exercise, it only remains to show that $\mathcal{A}_{AC}$ requires at most a polynomial number of steps in n, where n the size parameter for problem A. To see this, first note that the non-query steps of $\mathcal{A}_{AC}$ are the same non-query steps as $\mathcal{A}_{AB}$ and $\mathcal{A}_{BC}$. By assumption, $\mathcal{A}_{AB}$ requires at most $p(n)$ steps, for some polynomial $p(n)$, and $\mathcal{A}_{BC}$ requires at most $q(m)$ steps for some polynomial $q(m)$, where m is the size parameter for B. Also, since $\mathcal{A}_{AB}$ makes at most $p(n)$ queries (why?) to the B-oracle, it follows that $\mathcal{A}_{AC}$ calls the $\mathcal{A}_{BC}$ function at most $p(n)$ times. Moreover, the running time of each function call $\mathcal{A}_{BC}(y)$ is bounded by $q(m)$ and thus the m parameter is bounded by $p(n)$, since $p(n)$ is the maximum

number of allowable steps that can be made to construct a query to the B-oracle, and we may assume that it takes a single algorithm step to construct a single bit of query y. Thus, the execution of a function call to function $\mathcal{A}_{BC}$ will have a running time that is bounded by $q(p(n))$, which is a polynomial, since the compostion of two polynomials is also a polynomial. Finally, since there are at most $p(n)$ function calls to $\mathcal{A}_{BC}$, it follows that the total running time due to the function calls is bounded by the polynomial $p(n)q(p(n))$, and so $\mathcal{A}_{AC}$ has running time

$$\mathrm{O}(p(n) + p(n)q(p(n))) = \mathrm{O}(p(n)q(p(n))),$$

which is a polynomial. Therefore, $A \leq_{\mathrm{T}}^{\mathrm{p}} C$.

6. **Basis step.** Suppose x is marked and has index 0. Then necessarily $x = u$ and so x is reachable from u.

   Inductive step. Assume that for some $i \geq 0$, any marked vertex w that has an assigned index of i is reachable from u. Show that any marked vertex with assigned index $i + 1$ is also reachable from u.

   Proving the inductive step. Let x be a marked vertex that has been assigned index $i + 1$. Let (w, x) be the edge responsible for the marking of x. Then w has assigned index i and by the inductive assumption is reachable from u. But then x is also reachable from u via a path from u to w, followed by traversing the edge (w, x).

   For the `Reachability` algorithm, use math induction to prove that any vertex x that gets marked is reachable from u. Hint: assign an index to each marked vertex that represents the distance of that vertex from u. In particular, assign u index 0. Then if vertex w has index i and (w, x) is the edge responsible for the marking of x, then assign x index $i + 1$. Perform the induction on the index assigned to vertex x.

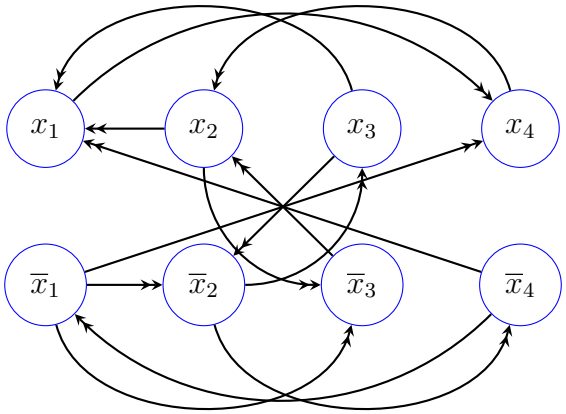
7. Queue sequence: $Q_1 = \{a\}$, $Q_2 = \{b, c\}$, $Q_3 = \{c, d, e, g\}$, $Q_4 = \{d, e, g, f\}$, $Q_5 = \{e, g, f\}$, $Q_6 = \{g, f\}$, $Q_7 = \{f, h\}$, $Q_8 = \{h\}$, $Q_9 = \emptyset$. Therefore, vertex k is not reachable from a since it was never marked and added to Q.

8. Satisfying assignment: $\alpha = (x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 1)$.

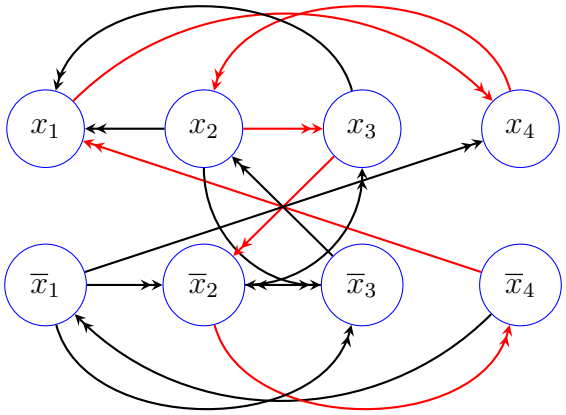
9. The satisfying assignment must assign $x_3 = 0$, since, based on the query answer, there is a path from x_3 to $\overline{x}_3$ which means the assumption that $x_3 = 1$ leads to a contradiction.

10. No, two queries at a minimum are needed. For example, what is the most one can say if the query `reachable`$(G_{\mathcal{C}}, x_3, \overline{x}_3)$ were answered "yes"? "no"?

11. The implication graph $G_{\mathcal{C}}$ is shown below. The reachability set for $l = x_1$ is $R = \{x_1, x_2, \overline{x}_3, x_4\}$ and α_R satisfies $\mathcal{C}$



12. See the following graph with inconsistent cycle highlighted in red.



13. First recursive case: $R_{x_1} = \{x_1, x_3\}$. Second recursive case:

$$R_{x_2} = \{x_2, x_5, \overline{x}_6, x_4, \overline{x}_2, \overline{x}_4, x_6, \overline{x}_5\}$$

is inconsistent. However, $R_{\overline{x}_2} = \{\overline{x}_2, \overline{x}_4, x_6, \overline{x}_5\}$ is consistent. Satisfying assignment: $\alpha = (x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 1)$.

14. $\alpha(x_3) = 1$, since R_{x_3} is consistent and $R_{\overline{x}_3}$ is inconsistent. Therefore, no satisfying assignment can assign 0 to x_3.

15. Maximum flow: $s(f) = 20$. See Figures 2 through 15. Note: in the residual networks, green edges are for the augmenting path, red for backward edges, and black for forward edges.

16. Maximum flow: $s(f) = 16$..

17. If H has a single vertex v, then, since v is flow-conserving and the edges entering/leaving H are exactly the edges entering/leaving v, we have

$$\sum_{e \in E^+(H)} f(e) = \sum_{e \in E^+(v)} f(e) = \sum_{e \in E^-(v)} f(e) = \sum_{e \in E^-(H)} f(e).$$

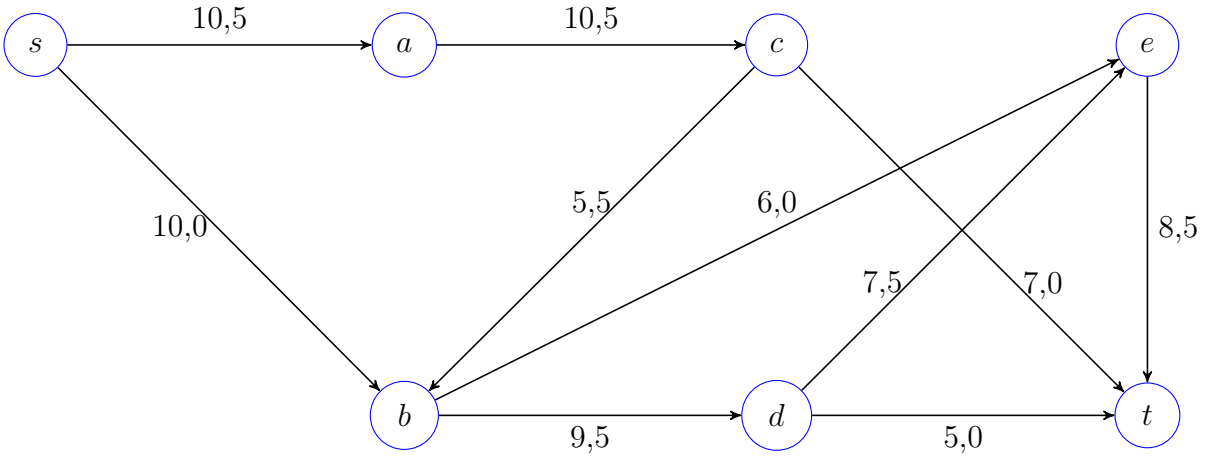


Figure 2: Network G with flow f_1

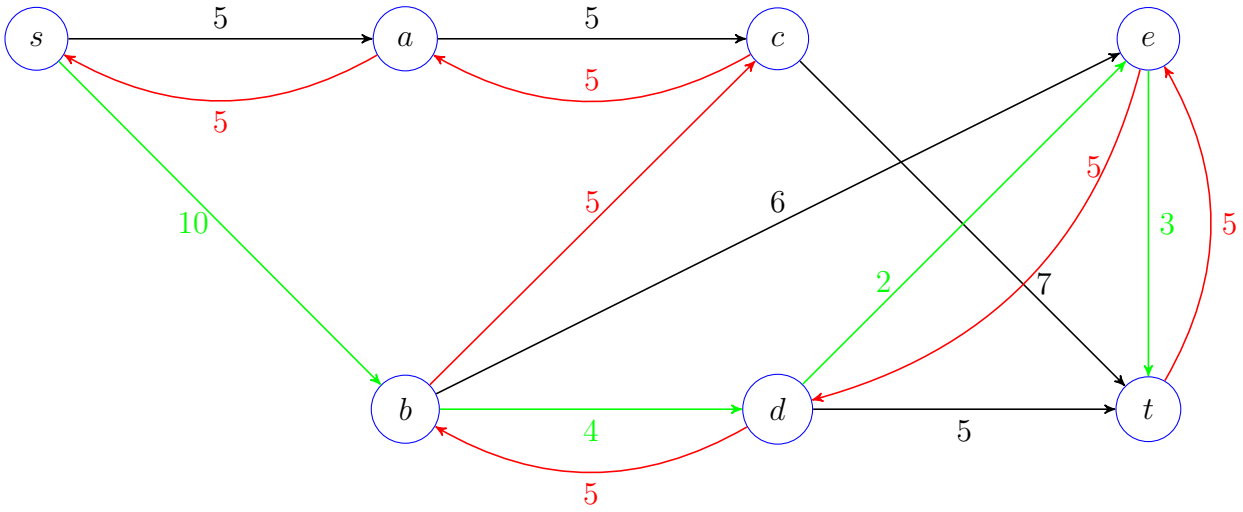


Figure 3: Residual Network G_{f_1} with P_1 in green and $\kappa = 2$

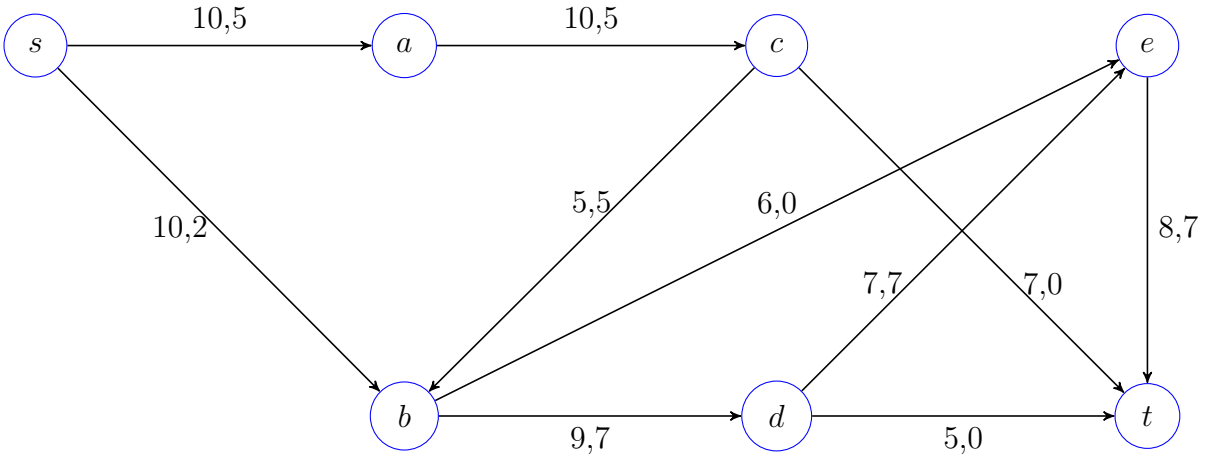


Figure 4: Network G with flow $f_2 = \Delta(f_1, P_1)$

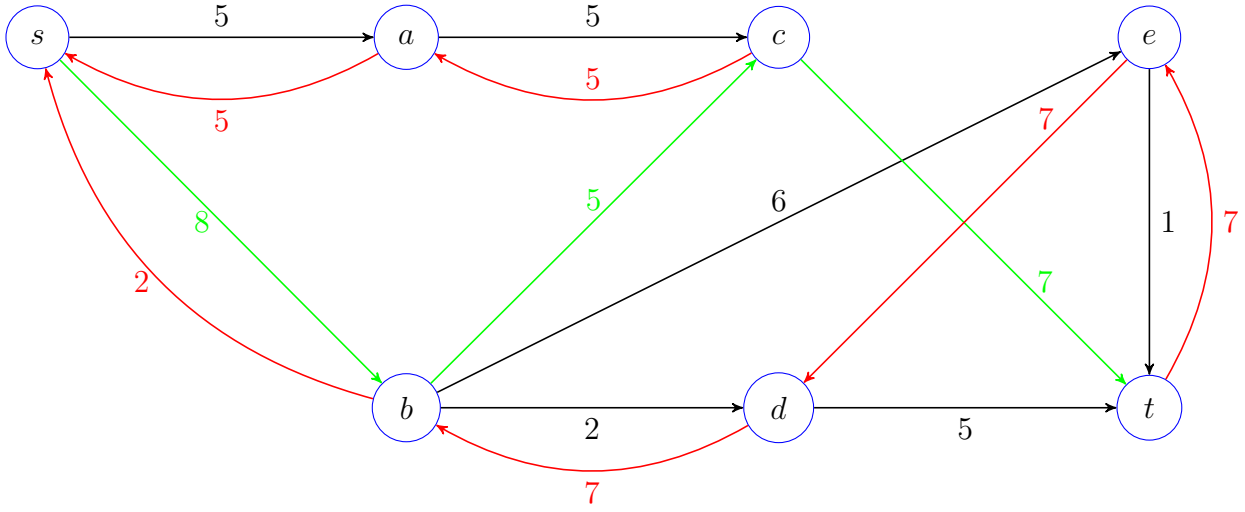


Figure 5: Residual Network G_{f_2} with P_2 in green and $\kappa = 5$

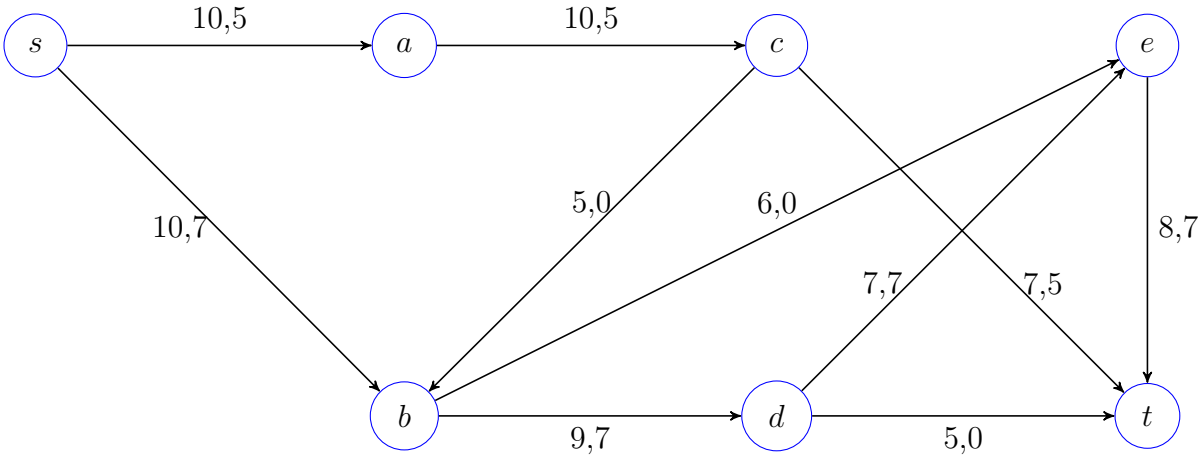


Figure 6: Network G with flow $f_3 = \Delta(f_2, P_2)$

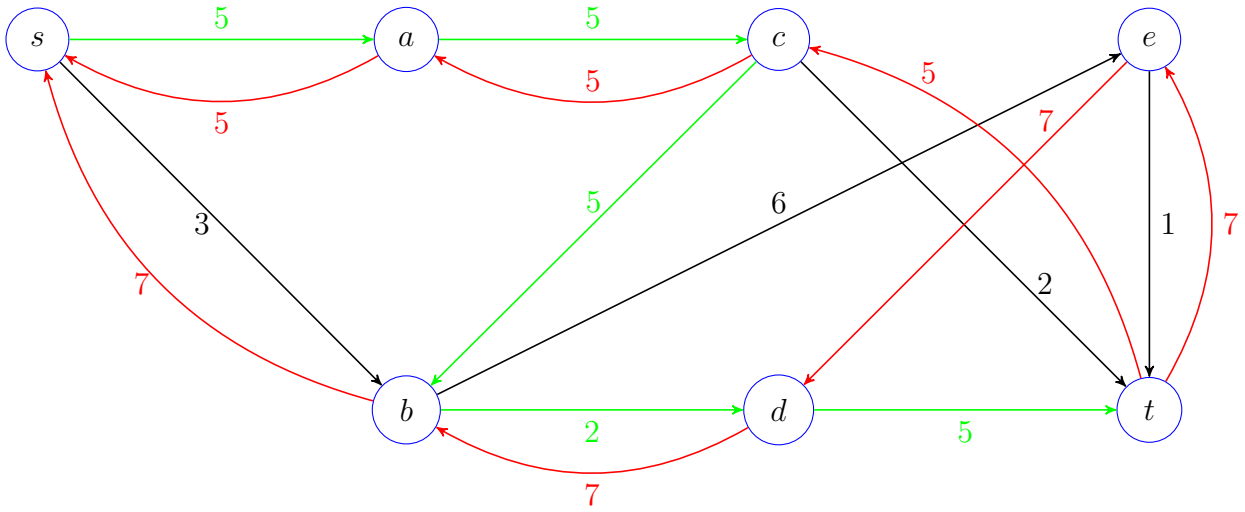


Figure 7: Residual Network G_{f_3} with P_3 in green and $\kappa = 2$. Note: this is an error since P_3 is NOT the longest path. I have not corrected it since it requires updating all subsequent figures)

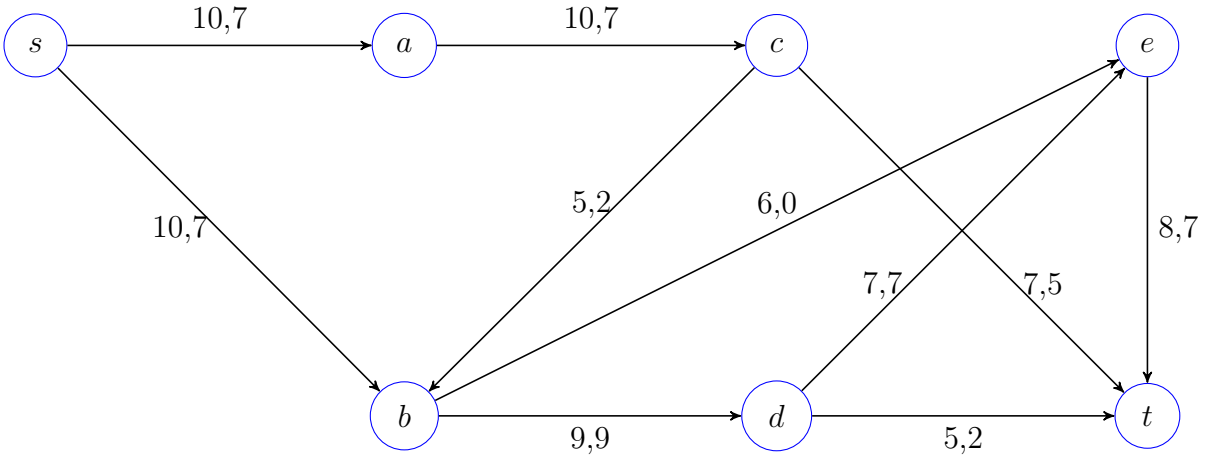


Figure 8: Network G with flow $f_4 = \Delta(f_3, P_3)$

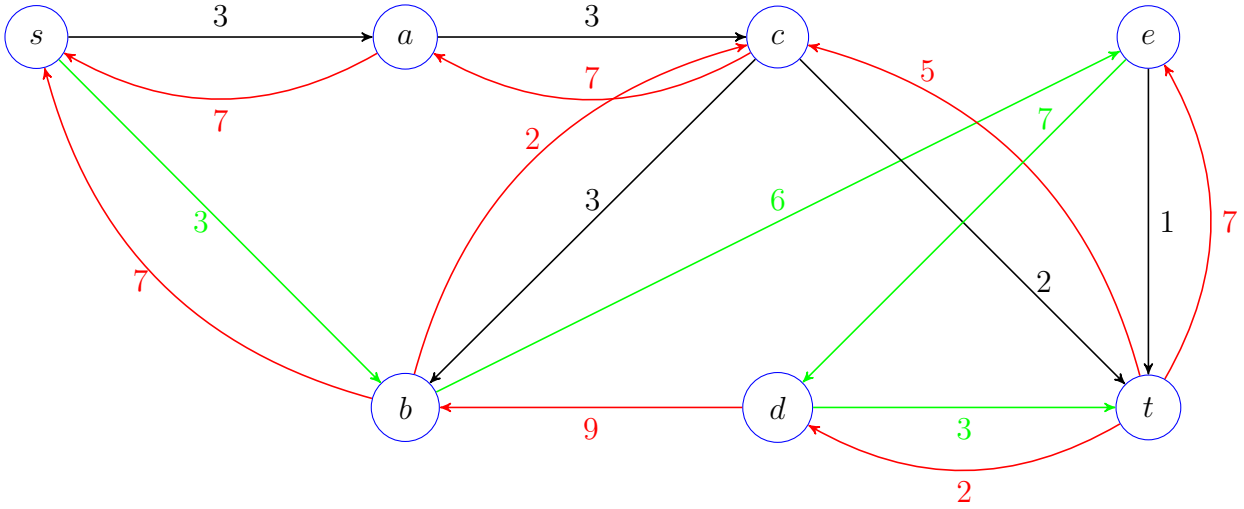


Figure 9: Residual Network G_{f_4} with P_4 in green and $\kappa = 3$ (Note: this is an error since P_4 is NOT the longest path. I have not corrected it since it requires updating all subsequent figures).

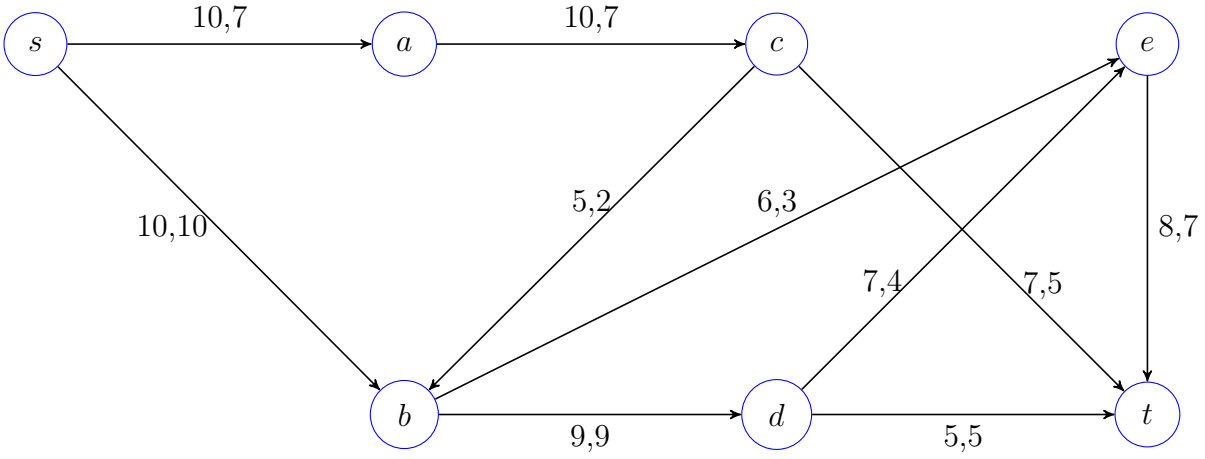


Figure 10: Network G with flow $f_5 = \Delta(f_4, P_4)$

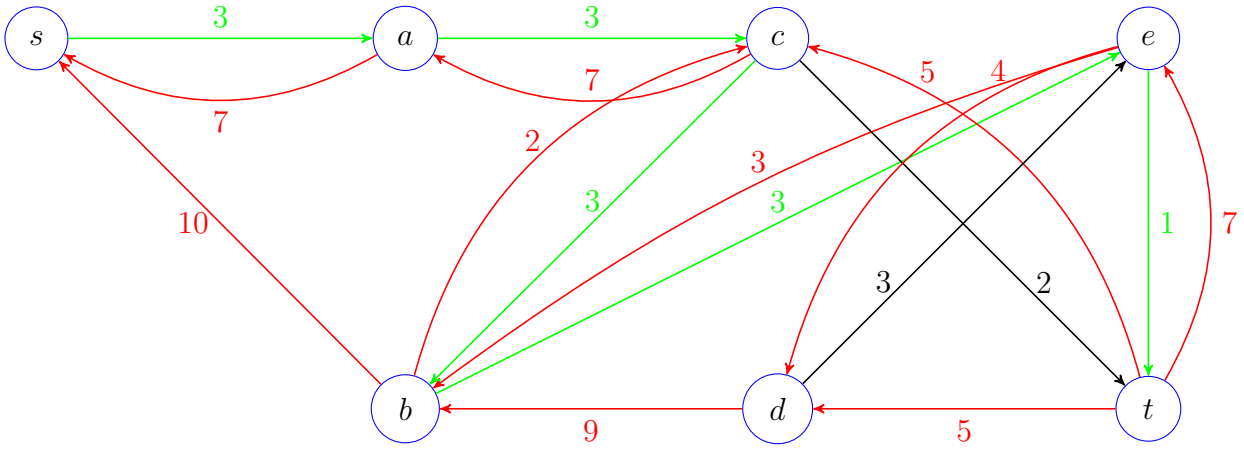


Figure 11: Residual Network G_{f_5} with P_5 in green and $\kappa = 1$

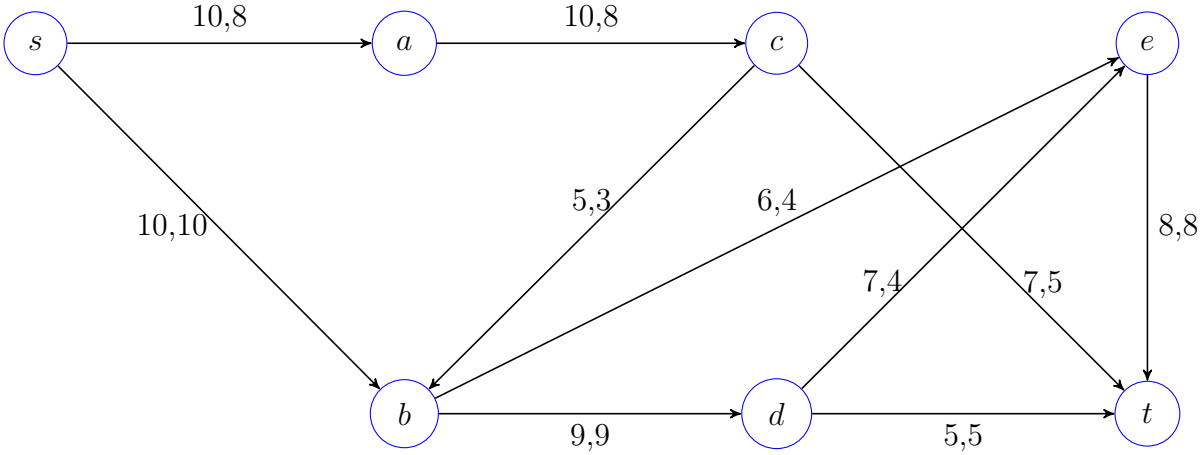


Figure 12: Network G with flow $f_6 = \Delta(f_5, P_5)$

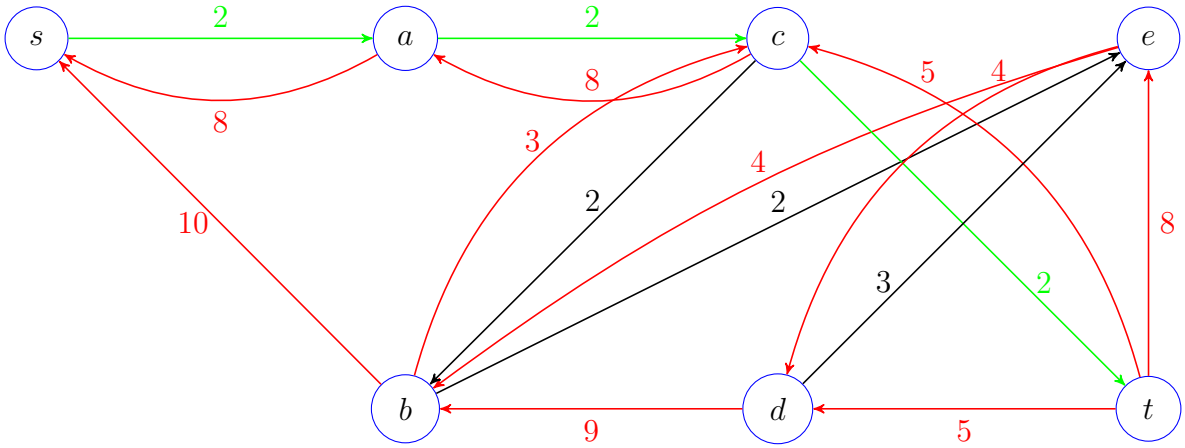


Figure 13: Residual Network G_{f_6} with P_6 in green and $\kappa = 2$

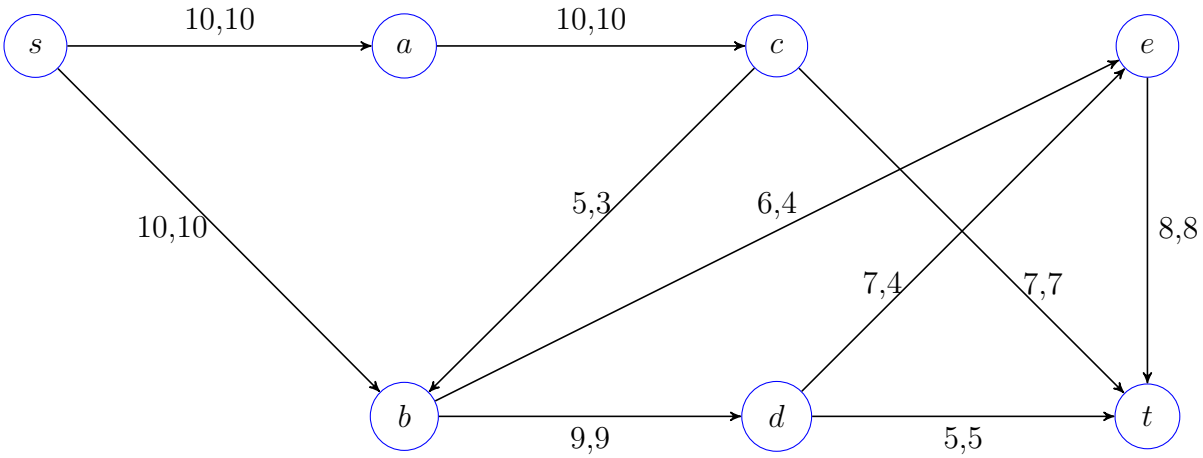


Figure 14: Network G with flow $f_7 = \Delta(f_6, P_6)$

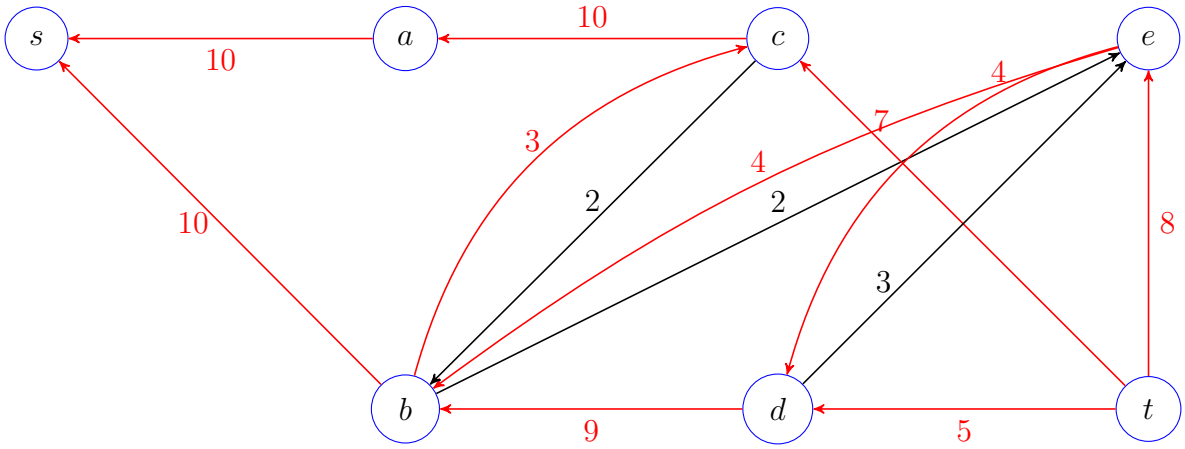


Figure 15: Residual Network G_{f_7} has no augmenting path from s to t. Algorithm terminates (it's about time!)

Now suppose it is true for any subgraph H with $n-1$ vertices, for some $n \geq 2$. Let v be an intermediate vertex that is external to H. Let a denote the sum of all flow entering v from vertices outside H, b denote the sum of all flow entering v from vertices within H, c denote the sum of all flow leaving v and entering vertices outside H, and d denote the sum of all flow leaving v and entering vertices within H. Thus, by conservation of flow through v, we have $a + b = c + d$. Let F denote the total flow entering H. By the inductive assumption, F also equals the total flow leaving H. Now suppose v is added to H to make a flow-consering subgraph having n vertices. Then the updated flow entering H is now $F + a - d$, since the outside flow enterng v is now entering H, while the flow entering H from v is now internal flow within H. Similarly, the flow leaving H is now $F - b + c$, since the flow from H to v is now internal flow, and the flow from v to vertices outside of H now counts as flow leaving H. Therefore, H remains flow conserving iff $F + a - d = F - b + c$ iff $a + b = c + d$, which is true, and the result is proved.

18. Suppose $V - \{v_0\}$ is a set of flow-conserving vertices, and consider the subgraph H whose vertex set is $V - \{v_0\}$. Since the vertices of H are all flow conserving, it follows from the previous exercise that the flow entering H must equal the flow leaving H. But there is only one vertex outside of H, namely v_0 and it has zero in-degree. Therefore, there is 0 flow leaving H which contradicts H being flow conserving. Therefore, H must have at least one vertex that do not conserve flow, and so G has ast least two vertices that do not conserve flow.

19. Consider an algorithm that is almost identical to the reachability algorithm described in Section 3. It begins by placing source vertex s into an initially-empty queue Q. Moreover, in the corresponding `while` loop, when vertex w is removed and if there is an edge $e = (w, x)$ for which i) $f(e) > 0$ and ii) x is unmarked, then x is marked and entered into Q. Let H denote the set of all vertices that are marked by the algorithm, excluding s. By the previous exercise, there must be at least one vertex in H that does not conserve flow. But the only possible vertex having this property is t. Therefore, t gets marked and so there must be a path from s to t whose traversed edges all have positive flow.

20. Let $e \in E$ be a forward edge of the network G_f. **Case 1:** $g(e) = f(e)$. In this case $\delta(e) = \delta(e^r) = 0$ and so the capacity limits are not exceeded. **Case 2:** $f(e) > g(e)$, then $\delta(e) = 0$, while $\delta(e^r) = f(e) - g(e) > 0$. Moreover,

$$\delta(e^r) = f(e) - g(e) \leq f(e)$$

which is the capacity of e^r in G_f. **Case 3:** $f(e) < g(e)$. Then $\delta(e^r) = 0$, while $\delta(e) = g(e) - f(e)$. Moreover,

$$\delta(e) = g(e) - f(e) \leq c(e) - f(e)$$

which is the capacity of e in G_f.

21. Consider a vertex $v \in V - \{s, t\}$, and let $e \in E$ be an edge entering v. **Case 1:** $g(e) = f(e)$. In this case $\delta(e) = \delta(e^r) = 0$. Notice that by placing $f(e)$ amount of flow through e^r (which is leaving v) and $g(e)$ amount through e, we get the same net flow of 0. **Case 2:** $f(e) > g(e)$, then $\delta(e) = 0$, while $\delta(e^r) = f(e) - g(e)$. Again, notice that by instead placing $f(e)$ amount of flow through e^r and $g(e)$ amount through e, we get the same net flow of $g(e) - f(e)$. **Case 3:** $f(e) < g(e)$. Then $\delta(e^r) = 0$, while $\delta(e) = g(e) - f(e)$. Once more, by instead placing $f(e)$

amount of flow through e^r and $g(e)$ amount through e, we get the same net flow of $g(e) - f(e)$. Thus, the total δ-flow through G_f that is entering v can be computed as

$$\sum_{e \in E^+(v)} g(e) - \sum_{e \in E^+(v)} f(e).$$

In words, we obtain the total flow through G_f that is entering v by summing the total g-flow entering v and subtracting the total f-flow entering v. A similar argument shows that the total δ-flow through G_f that is leaving v is equal to

$$\sum_{e \in E^-(v)} g(e) - \sum_{e \in E^-(v)} f(e),$$

which is the total g-flow leaving v minus the total f-flow leaving v. The two expressions for total in-flow and total out-flow must be equal since f and g are both flows through G, and v is flow conserving with respect to any flow through G.