Turing Reducibility

Last Updated: August 28th, 2025

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 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 Computational Problems

Informally, when we think of a problem, we think of a situation that needs to be resolved (i.e. solved). Moreover, in computer science we think of a computing problem as having the following properties.

Definition 2.1. A **computing problem** is a collection of situations that share a common theme.

- Each situation is referred to as a **problem instance**, and represents a concrete example of the general problem.
- Each problem instance can be represented by a unique word over some alphabet, meaning that no two instances map to the same word. The word may then be encoded into a binary word so that the problem instance can be stored in computer memory.

Three types of problems

Decision The solution to a problem instance is either Yes or No, equivalently True (1) or False (0). An instance x for which the solution is 1 (respectively, 0) is called a **positive instance** (respectively, **negative instance**).

Optimization The solution to a problem instance x is a number that represents the greatest (or least) quantity of some entity that is associated with x

Miscellaneous Any computation problem that is neither a decision nor an optimization problem

Example 2.2. For each of the following problems, determine if it is a decision, optimization, or miscellaneious problem.

a. An instance of the `Prime` problem is a natural number $n \geq 0$, and the problem is to decide if n is a prime number, meaning that its only natural divisors are 1 and n. Decision Problem

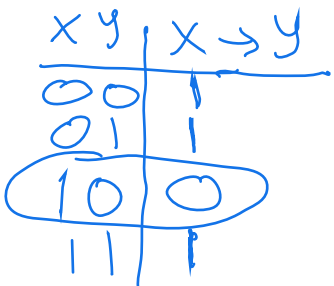
b. An instance of the `Maximum Subsequence Sum (MSS)` is a sequence (array) a of integers. The problem is to determine the greatest sum that can be made by any subsequence of a. For example, determine the maximum subsequence sum for any subsequence of

Optimization $3, -4, 2, 5, -2, -4, 0, 2, 3, -2, 1.$

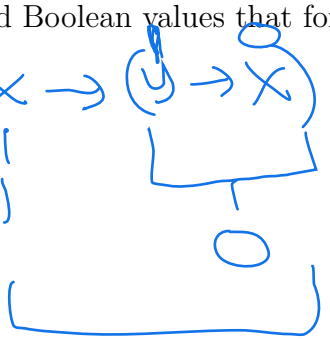
c. An instance of `Sort` is an array a of integers. The problem is to produce another array b whose members are the members of a, but in sorted order. Miscellaneous

d. An instance of `Fallible` is a Boolean formula F. Is there an assignment that can be made to the variables of F so that F evaluates to 0? For example, given the logical formula $F(x, y) = x \to (y \to x)$, can x and y be assigned Boolean values that force F to evaluate to 0?

$F(x,y) = x \to (y \to x)$



Decision Problem



Contradiction
x cannot be simultaneously 1 and 0

∴ F is a tautology negative instance of Fallible

R → W
a e
i t
n
s

3 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 3.1. 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;
}
```

call to an add-oracle

In Example 3.1, 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 need to know how the oracle is providing its answers. In case of the `multiply` function in Example 3.1, 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 3.2. 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.

Mult $\leq_T$ Add

Note that $\leq_T$ is a **relation** over the set of computing problems, in that, for any two problems A and B, $A \leq_{\mathrm{T}} B$ evaluates to 0 or 1.

Definition 3.3. 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.

Example 3.4. Is it always true that $A \leq_T A$ for any problem A? In other words, is $\leq_T$ a **reflexive** relation? If $A \leq_T B$, is it always true that $B \leq_T A$? In other words, is $\leq_T$ a **symmetric** relation?

Solution. B : Halt. Instance: Program P, input x
Question: Does program P halt/terminate on input x?

A : Even. Instance: natural number $n \geq 0$.
Question is n even?

$A \leq_T B$ since Even is solvable by a program that makes zero queries to B.

$B \not\leq_T A$ since B is unsolvable and having an Even-oracle to query still makes B unsolvable.

Program for proving $A \leq_T A$ for any computational problem A.

```
input x   // x is an instance of A.
Return A(x) // query A-oracle about instance x.
```

4 Sets and Graphs Review

Our next example of Turing reducibility involves Turing reducing a logic problem to a graph problem. We now review some basic set and graph definitions that will be used throughout the remainder of the course.

4.1 Sets

Definition 4.1. A **set** represents a collection of items, where each item is called a **member** or **element** of the set.

List Notation the most common way to represent a set where the set members are listed one-by-one, and the list is delimited by braces. For example,

$$\{2, 3, 5, 7, 11\}$$

uses list notation to describe the set consisting of all prime numbers that do not exceed 11. Note that the order in which the members are listed does not matter. Indeed the sets $\{2, 3, 5, 7, 11\}$ and $\{3, 11, 5, 2, 7\}$ are identical. Also, each member occurs only *once* in the set, meaning that $\{1, 2, 2, 3, 3, 3\}$ is the same set as $\{1, 2, 3\}$. Note: a **multiset** is a set for which each member may occur more than once, but we will have little if any use for them in this course.

Informal List Notation uses **ellipsis** ... to indicate that a pattern is to be continued in the list, either indefinitely or up to some value.

Common Numerical Sets **Natural Numbers** $\mathcal{N} = \{0, 1, 2, \ldots\}$

Integers $\mathbb{I} = \{0, \pm 1, \pm 2, \ldots\} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

Rational Numbers or Fracations $\mathbb{Q} = \{p/q \mid p, q \in \mathbb{I} \wedge q \neq 0\}$

Empty Set the set having no members and denoted by $\emptyset$.

Membership Symbol $x \in A$ indicates that item x is a member of set A, while $x \notin A$ means that x is not a member of A.

Containment Symbol $A \subseteq B$ means that A is a **subset** of B. In other words, every member of A is also a member of B. Note: trivially, $\emptyset \subseteq B$ for every set B, but $A \subseteq \emptyset$ is only true when $A = \emptyset$.

Proper Containment Symbol $A \subset B$ means that A is a **subset** of B but that there is some member of B that is not a member of A. In this case we say that A is a **proper subset** of B.

Set Equality $A = B$ iff $A \subseteq B$ and $B \subseteq A$ are both true statements.

Set Cardinality $|A|$ denotes the number of items of A. We also refer to $|A|$ as the size of A. Note: $|\mathcal{N}| = |\mathbb{I}| = |\mathbb{Q}| = \infty$ since the members of all three sets can be placed in an infinite list.

Example 4.2. The following are all true statements.

The following are all true statements.

a. $61 \in \{2, 3, 5, 7, 11, \ldots, 97\}$

b. $39 \notin \{2, 3, 5, 7, 11, \ldots, 97\}$

c. $\{3\} \in \{\emptyset, \{1\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 2, 3\}\}$

d. $3 \notin \{\emptyset, \{1\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 2, 3\}\}$

e. $\{7, 23, 59\} \subset \{2, 3, 5, 7, 11, \ldots, 97\}$ and $\{7, 23, 59\} \subseteq \{2, 3, 5, 7, 11, \ldots, 97\}$

f. $\{3\} \not\subset \{\emptyset, \{1\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 2, 3\}\}$

g. $|\emptyset| = 0$ and $|\{\emptyset, \{1\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 2, 3\}\}| = 6$

□

4.2 Graphs

A **Graph** is a pair of sets V and E, where

Graph $G = (V, E)$ V is a set of **vertices**, also called **nodes**, while E is a set whose members are pairs of vertices and are called **edges**. Each edge may be written as a tuple of the form (u, v), where $u, v \in V$.

Adjacency and Incidence If $e = (u, v)$ is an edge, then we say that u is **adjacent** to v, and that e is **incident** with u and v.

Order $|V| = n$ is called the **order** of G.

Size $|E| = m$ is called the **size** of G.

Path A **path** P of length k in a graph is a sequence of vertices $P = v_0, v_1, \ldots, v_k$, such that $(v_i, v_{i+1}) \in E$ for every $0 \leq i \leq k - 1$.

Simple Path $P = v_0, v_1, \ldots, v_k$, where the vertices $v_0, v_1, \ldots, v_k$ are all distinct.

Path Length represents the number of *edges* that are traversed when following the path. If $P = v_0, v_1, \ldots, v_k$, then the length of P equals k and we write $|P| = k$.

Cycle A **cycle** is a path that begins and ends at the same vertex and has a length of at least 3.

Degree The **degree** of a vertex v, denoted as $\deg(v)$, equals the number of edges that are incident with v. Note: loop edges are counted twice.

Example 4.3. Let $G = (V, E)$, where

$$V = \{SD, SB, SF, LA, SJ, OAK\}$$

is a set of California city abbeviations, and

$$E = \{(SD, LA), (SD, SF), (LA, SB), (LA, SF), (LA, SJ), (LA, OAK), (SB, SJ)\}$$

are edges, each of which represents the existence of one or more flights between two cities that some airline provides. Figure 1 shows a graphical representation of G. G has order 6 and size 7.

Figure 2 shows a simple path of length 4. Figure 3 shows a cycle of length 3. Let's verify the Handshaking theorem which states that the sum of all vertex degrees equals twice the number of edges.

$$\deg(\text{SF}) + \deg(\text{LA}) + \deg(\text{SD}) + \deg(\text{OAK}) + \deg(\text{SJ}) + \deg(\text{SB}) =$$

$$2 + 5 + 2 + 1 + 2 + 2 = 14 = 2 \cdot 7 = 2|E|.$$

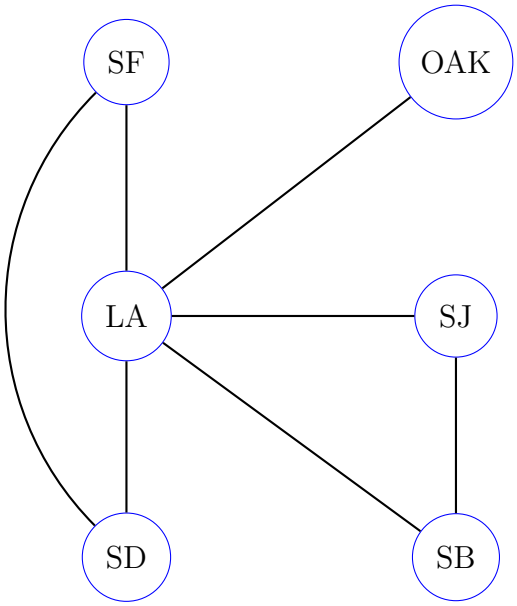


Figure 1: Graphical Representation of G

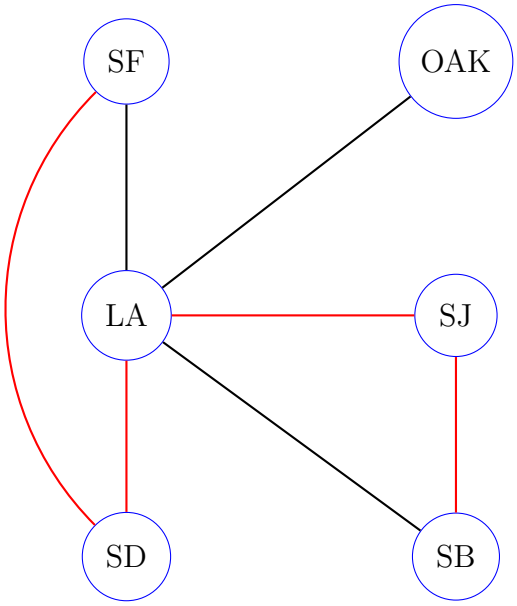


Figure 2: Simple path (in red) P = SF,SD,LA,SJ,SB of length 4

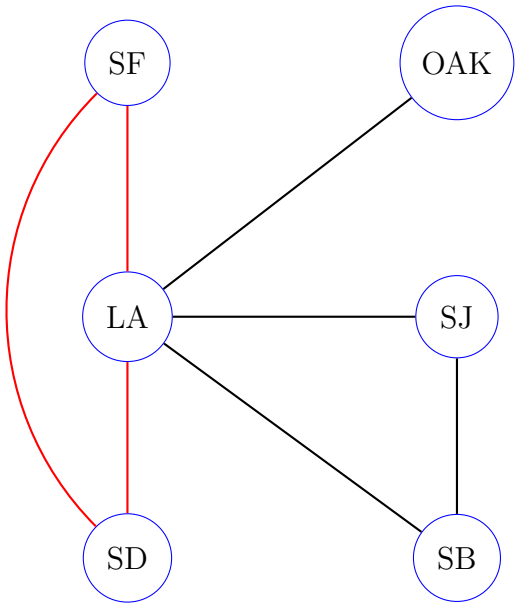
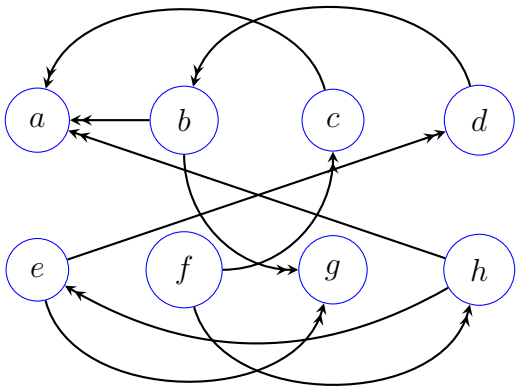


Figure 3: Cycle (in red) $C = \text{SF,SD,LA,SF}$ of length 3

Definition 4.4. A **directed graph** $G = (V, E)$ is a graph for which each edge $e = (u, v)$ of G is oriented so that, when traversing e, u must always come before v. In other words, directed edge e is like a one-way street where traffic must travel from u to v.

Example 4.5. Directed graphs play an important role in planning and scheduling. The vertices of the graph below represent different tasks that must be completed for a project. A directed edge (u, v) means that the completion task u is necessary before task v may be started. In what order(s) may you complete these tasks in order to complete the project? What is the longest path in G?



5 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 5.1. 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 `Reachability` problem is an example of a **decision problem**, i.e. the solution to each problem instance is either 0 or 1. A problem instance whose solution is 1 is said to be a **positive instance**. Otherwise it is called a **negative instance**. Any algorithm that solves a decision problem is said to **decide** the problem.

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 5.2. 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 $O(2m+n) = O(m+n)$. □

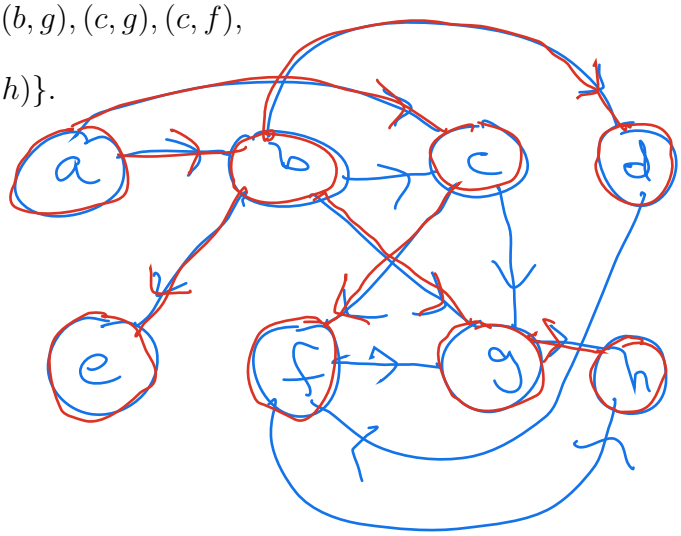
Example 5.3. Show the contents of the queue Q during the execution of the above algorithm on the graph $G = (V, E)$, where

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

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.



1. $Q = a$
2. $Q = b, c$
3. $Q = e, d, e, g$
4. $Q = d, e, g, f$
5. $Q = e, g, f$
6. $Q = g, f$
7. $Q = f, h$
8. $Q = h$
9. $Q = \emptyset$

— Breadth-First Tree formed by algorithm

Conclusion: h is reachable from a since h was added to Q

6 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 equals $\{0, 1\}$. 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

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

7 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 7.1. 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 7.2. 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 7.3. Provide a satisfying assignment for

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

$\alpha = (x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1)$

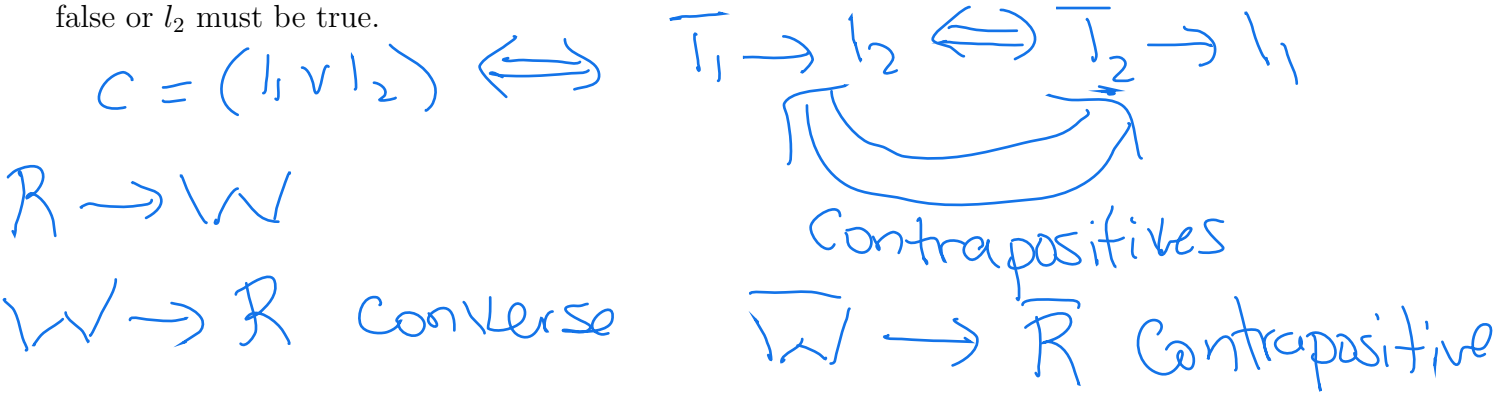
We now demonstrate how to Turing reduce `2SAT` to `Reachability`, meaning that we can solve an instance of `2SAT` by making queries to a `Reachability`-oracle.

Definition 7.4. 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.

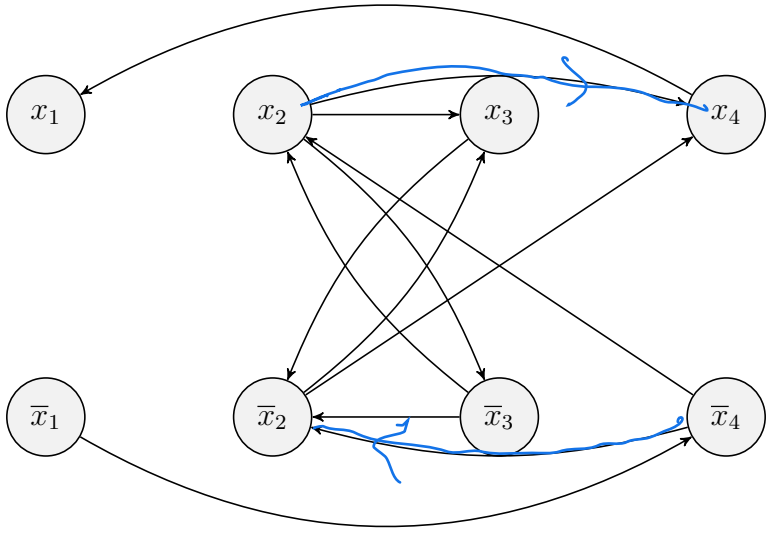


Example 7.5. Verify that $l_1 \vee l_2$ is logically equivalent to both $\overline{l_1} \to l_2$ and $\overline{l_2} \to l_1$.

Solution. Exercise

Example 7.6. 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 7.7. Given implication graph $G_{\mathcal{C}}$ and path

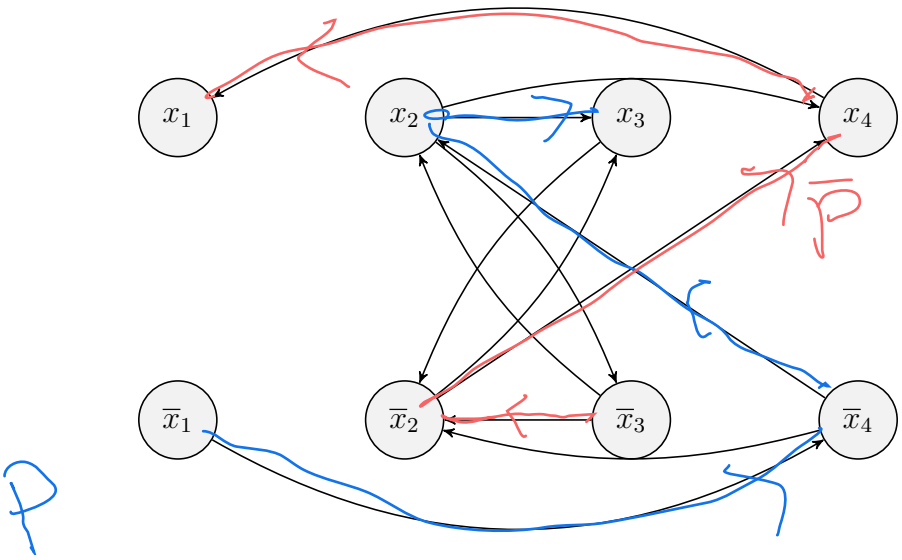
$$P = l_1, \ldots, l_n,$$

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 $(\overline{y}, \overline{x})$, since both are logically equivalent to $\overline{x} \vee y$. Thus, every edge in a path $P = l_1, l_2, \ldots, l_{n-1}, l_n$ corresponds with its contrapositive in the path $\overline{P} = \bar{l}_n, \bar{l}_{n-1}, \ldots, \bar{l}_2, \bar{l}_1$, and thus both P and $\overline{P}$ are paths of $G_{\mathcal{C}}$. □

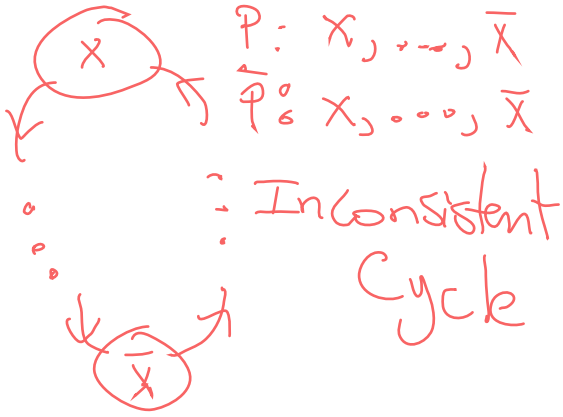
Example 7.8. Given the path $P = \overline{x}_1, \overline{x}_4, x_2, x_3$ with respect to the implication graph $G_{\mathcal{C}}$ below, verify that $\overline{P}$ is also in $G_{\mathcal{C}}$.



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

Theorem 7.9. `2SAT` instance $\mathcal{C}$ is satisfiable iff every cycle in $G_{\mathcal{C}}$ is **consistent** meaning that the set of literals that make up the cycle is consistent.

Before proving Theorem 7.9, 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 $G_{\mathcal{C}}$ has only consistent cycles iff, for every variable x, either x is not reachable from $\overline{x}$, or $\overline{x}$ is not reachable from x (or both). For each x, this 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`.

Query to a Reachability Oracle

Example 7.10. Consider a 2SAT instance $\mathcal{C}$ for which $G_{\mathcal{C}}$ has the inconsistent cycle

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

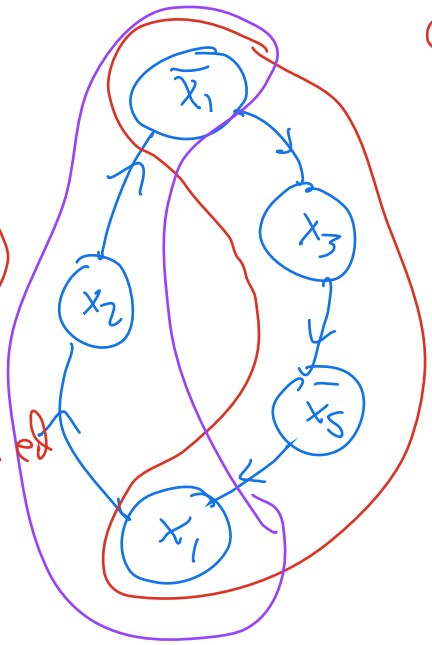
Verify that $\mathcal{C}$ is unsatisfiable.

Associated clauses:

$(\overline{x}_1 \vee x_2)$, $(\overline{x}_2 \vee \overline{x}_1)$

$\alpha(x_1) = 1$ does not satisfy these clauses

unsatisfied



Associated clauses:

$(x_1 \vee x_3)$, $(\overline{x}_3 \vee \overline{x}_5)$, $(x_5 \vee x_1)$

not satisfied

$\alpha(x_1) = 0$ does not satisfy those clauses

We can generalize the previous example by stating the following proposition.

Proposition 7.11. Given a 2SAT instance $\mathcal{C}$ that includes variable x, the following statements are true.

1. If there is a path from x to $\overline{x}$ then no assignment that assigns $x = 1$ can satisfy $\mathcal{C}$.
2. If there is a path from $\overline{x}$ to x then no assignment that assigns $x = 0$ can satisfy $\mathcal{C}$.

Proposition 7.14 below 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.

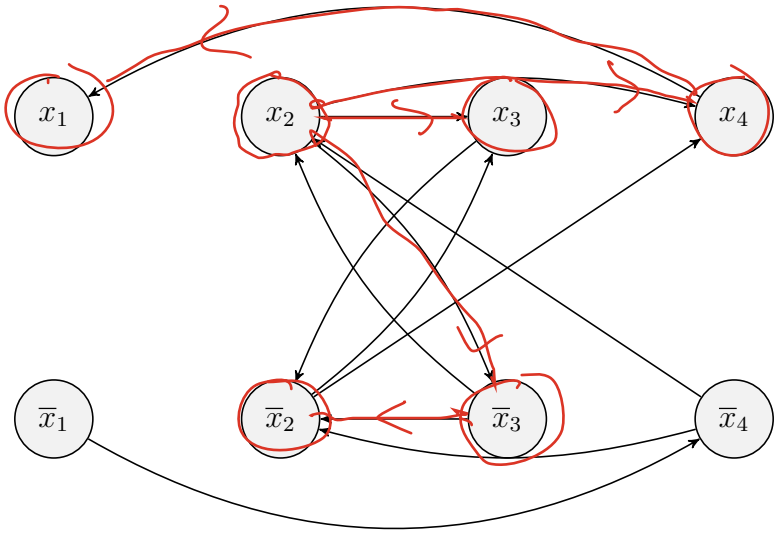
Definition 7.12. Let $G = (V, E)$ be a graph and $v \in V$ a vertex of G. Then the **reachability set** of v in G is the set of all vertices that can be reached by v along some path (regardless of its length).

Example 7.13. Verify that the reachability set for vertex x_2 of the implication graph in Example 7.6 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?

inconsistent



Proposition 7.14. Given 2SAT instance $\mathcal{C}$, implication graph $G_{\mathcal{C}}$, and vertex/literal l, if R_l, the following statements are true.

1. If R_l is an inconsistent set of literals, then no assignment that assigns $l = 1$ can satisfy $\mathcal{C}$.
2. If R_l is a consistent set of literals, then 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.

A. Since R_l is inconsistent, There is a variable x for which both x and $\overline{x}$ belong to R_l.

B. Let $P = l, l_1, \ldots, l_r, x$ be a path from l to x.

C. Let $Q = l, l'_1, \ldots, l'_s, \overline{x}$ be a path from l to $\overline{x}$.

D. Then $P \circ \overline{Q} = l, l_1, \ldots, l_r, x, \overline{l'_s}, \ldots, \overline{l'_1}, \overline{l}$ is a path from l to $\overline{l}$.

E. Therefore, by Propostion 7.11, no assignment that assigns $l = 1$ can satisfy $\mathcal{C}$. □

Proof of Statement 2.

A. Without loss of generality, we assume that x is a variable that is reachable from l (a similar argument can be made by assuming $\overline{x}$ is reachable from l).

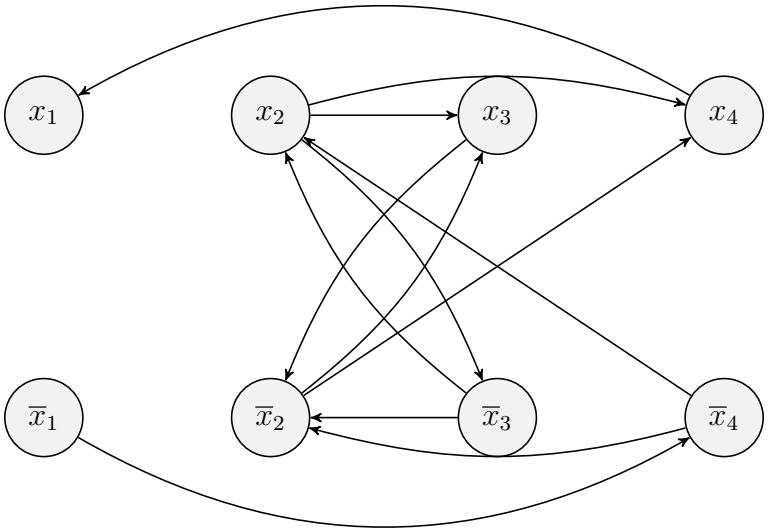
B. Consider any clause $c = (x, l')$, where l' is some literal. Then, since $x \in R_l$, $\alpha_{R_l}(x) = 1$ and c is satisfied.

C. Now consider any clause $c = (\overline{x} \vee l')$, where again l' is some literal. Notice that (x, l') is an edge of $G_{\mathcal{C}}$. Hence, since x is reachable from l, l' is also reachable from l. Thus, since $l' \in R_l$, $\alpha_{R_l}(l') = 1$ and c is satisfied.

D. Therefore, so long as a clause c has either a variable or its negation that is reachable from l, then c will be satisfied by α_{R_l} and α_{R_l} represents a **partial satisfying assignment** for $\mathcal{C}$. □

Example 7.15. 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 $R_{\overline{x}_1}$ is an inconsistent reachability set which means that no assignment with $x_1 = 0$ can satisfy $\mathcal{C}$.

Exercise



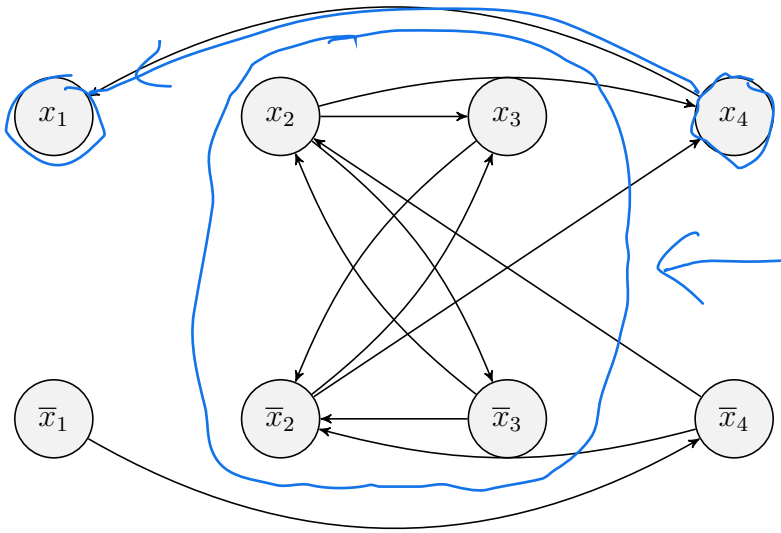
Example 7.16. Verify that the reachability set for vertex x_4 of the implication graph in Example 7.6 is equal to the consistent set

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

and show that $\alpha_{R_{x_4}}$ satisfies all clauses of $\mathcal{C}$ that use one of the variables from R_{x_4}.

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)	$\overline{x}_2 \to x_4$	$\overline{x}_4 \to x_2$
$(x_1, \overline{x}_4)$	$\overline{x}_1 \to \overline{x}_4$	$x_4 \to x_1$

Remain to be satisfied



Proof of Theorem 7.9. The statement of the theorem is of the form $P \leftrightarrow Q$, where

1. P stands for "$\mathcal{C}$ is satisfiable"
2. Q stands "$G_{\mathcal{C}}$ has only consistent cycles".
3. Thus, We can thus prove the two mathematical statements: $P \to Q$ and $Q \to P$.

Proving $P \to Q$

1. We use an indirect proof by proving the contrapositive of $P \to Q$, namely $\overline{Q} \to \overline{P}$.
2. Assume $\overline{Q}$: $G_{\mathcal{C}}$ has at least one inconsistent cycle.
3. Then there is a variable x in the cycle for which there is a path from x to $\overline{x}$ as well as a path from $\overline{x}$ to x.
4. Then both R_x and $R_{\overline{x}}$ are inconsistent reachcability sets.
5. Hence, by Proposition 7.14, there is no truth assignment that will satisfy $\mathcal{C}$.
6. Therefore, $\overline{P}$ is true namely that $\mathcal{C}$ is unsatisfiable. □

Proof of Theorem 7.9 Continued.

Proving $Q \to P$

1. Assume Q: $G_{\mathcal{C}}$ has only consistent cycles.
2. To prove P : "$\mathcal{C}$ is satisfiable" we use mathematical induction on the number of variables n that appear in one or more of the clauses of $\mathcal{C}$.
3. **Basis Step.** $n = 0$, i.e. $\mathcal{C} = \emptyset$. Then $\mathcal{C}$ is satisfiable. Why? It helps to reframe the definition of satisfiability as "there is some assignment α for which no clause is unsatisified by α". This definition is equivalent to the one given Definition 7.2 when $\mathcal{C} \neq \emptyset$. Moreover, using this restated definition, $\mathcal{C} = \emptyset$ implies that $\mathcal{C}$ is satisfiable since the empty assignment $\alpha = (\ )$ is a variable assignment for which no clause of $\mathcal{C}$ is unsatisfied by α (because $\mathcal{C}$ has no clauses!).
4. **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 $n \geq 1$. Let $\mathcal{C}$ be a `2SAT` instance with n variables and only consistent cycles. We show that $\mathcal{C}$ must also be satisfiable.
5. Let x be a variable of $\mathcal{C}$. Then by assumption is must be true that either there is no path from x to $\overline{x}$ in $G_{\mathcal{C}}$ or there is no path from $\overline{x}$ to x. Without loss of generality, assume that there is no path from x to $\overline{x}$.
6. Then R_x must be a consistent set of literals. Otherwise, as was shown in the proof of Proposition 7.14, $\overline{x}$ would be a member of R_x which we've assumed is not the case.
7. Moreover, Proposition 7.14 also implies that α_{R_x} satisfies every clause that has a variable that gets assigned by α_{R_x}, including x.
8. Let C_{R_x} denote the set of clauses satisfied by α_{R_x} and consider the new `2SAT` instance $\mathcal{C}' = \mathcal{C} - C_{R_x}$ that is the result of removing the clauses in C_{R_l} from $\mathcal{C}$.
9. 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.
10. Hence, by the inductive assumption, $\mathcal{C}'$ is satisfiable.
11. Let α' denote a satisfying assignment for $\mathcal{C}'$, then $\alpha' \cup \alpha_{R_x}$ satisfies $\mathcal{C}$.
12. Therefore $\mathcal{C}$ is satisfiable. □

The proof of Theorem 7.9 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. In the algorithm we assume that the variables appearing in the root problem are indexed as $x_1, \ldots, x_n$, for some $n \geq 1$.

Improved 2SAT Algorithm

Name: `improved_2sat`

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 $G_{\mathcal{C}}$.

Let i be the least index for which x_i is a variable of $\mathcal{C}$.

$S \leftarrow \emptyset$ //S is the desired set of consistent literals and initialized as empty

If R_{x_i} is a consistent reachability set, then $S \leftarrow R_{x_i}$.

If $S = \emptyset$ and $R_{\overline{x}_i}$ is a consistent reachability set, then $S \leftarrow R_{\overline{x}_i}$.

If $S = \emptyset$

$\alpha \leftarrow \emptyset$.

Return `false`.

//$S \neq \emptyset$

Update α: $\alpha \leftarrow \alpha \cup \alpha_S$.

$\mathcal{C}' \leftarrow \mathcal{C} - C_S$, where C_S denotes all clauses satisfied by α_S.

Return `improved_2sat`$(\mathcal{C}', \alpha)$.

Notice that the algorithm requires $O(m + n)$ steps since every edge of $G_{\mathcal{C}}$ is traversed at most once when using the reachability algorithm to compute either R_{x_i} or $R_{\overline{x}_i}$.

Example 7.17. Use the improved 2SAT algorithm to determine a satisfying assignment for

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

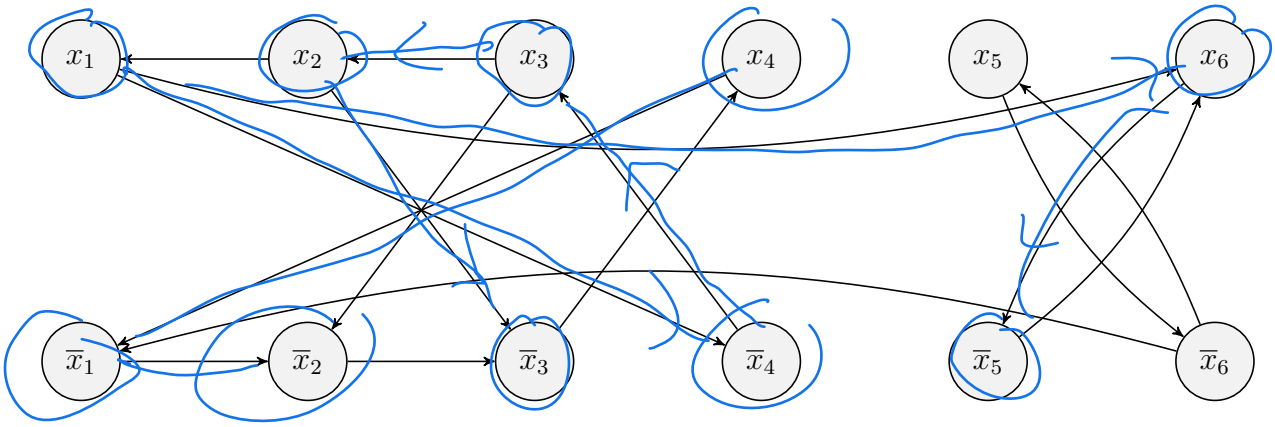
Start off by choosing $l = x_1$.

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



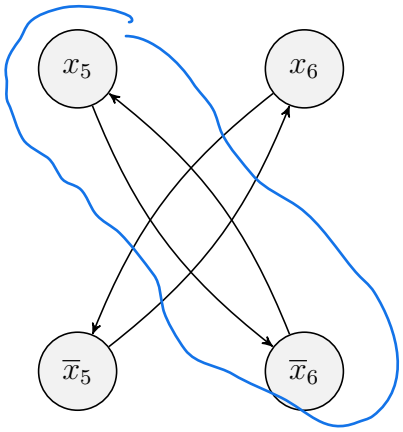
3. Compute $R_{x_1} = \{x_1, \overline{x}_1, x_2, \overline{x}_2, x_3, \overline{x}_3, x_4, \overline{x}_4, \overline{x}_5, x_6\}$ which is inconsistent.

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:

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

8 Finding Maximum Network Flows

Network Flow

A **transportation network** is directed graph $G = (V, E, c, s, t)$, 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**. The idea of a transportation network is that a resource (e.g. water, oil, electricity, data, etc.) is to be sent from the source vertex s to the destination vertex t. Moreover, the resource is moved from s through the edges of G on its way to t. Finally, the capacity of an edge e reflects a bound of how much of the resource can be sent through e at any given time. Perhaps the most example is in the case when the resource is water and the edges are pipes, and a pipe capacity is proportional to its diameter.

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:

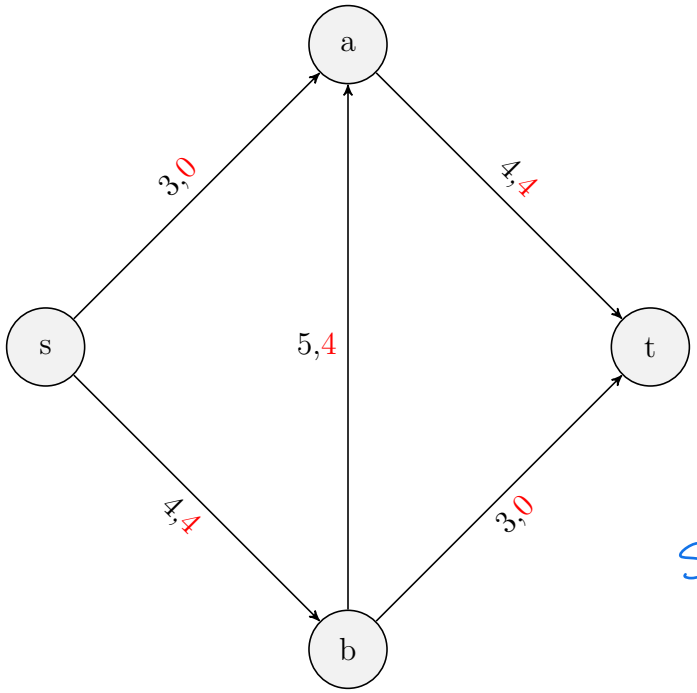
1. For every $e \in E$, $f(e) \leq c(e)$. In other words, the flow through an edge should not exceed the edge's capacity.

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 8.1. For the transortation network below, the left edge label is the edge capacity, while the right edge-label is the flow value for a flow f.



$S(f) =$
size of flow $= 4$

The **size** of a flow f through a network, denoted $|f|$, is defined as

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

and represents the total flow that is leaving source s. For the flow f in Example 8.1 we have $|f| = 0 + 4 = 4$.

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 $|f| \geq |f'|$ for all other flows f'.

8.1 Minimum cuts

One means of bounding the size of a flow that can be defined on a transportation network $G = (V, E, c, s, t)$ is to examine a **cut** of the network which consists of two disjoint sets of vertices X and Y for which i) $X \cup Y = V$ ii) $s \in X$, and iii) $t \in Y$. The **size** of the cut is defined as

$$c(X, Y) = \sum_{e \in X \times Y} c(e).$$

In words, it is the sum of the capacities of all edges that start in X and end in Y. Note that if $e = (x, y) \in X \times Y$ is not in E, then $c(e)$ is defined as 0. A cut is said to be a **minimum cut** of G iff its size is the least of all possible cuts of G.

Proposition 8.2. Given network $G = (V, E, c, s, t)$, flow f over G, and cut (X, Y), it must be the case that

$$|f| \leq c(X, Y).$$

Corollary 8.3. Given network $G = (V, E, c, s, t)$, a flow over G of maximum size cannot exceed the size of a minimum cut of V.

In the next section we prove something stronger, namely that, if f is a maximum flow over network G, then there is a cut (X, Y) of G for which

$$|f| = c(X, Y),$$

which implies that the size of a maximum flow for a network must equal the size of a minimum cut for the network.

The proof of Proposition 8.2 relies on the following intuitive fact of transportation networks (Exercise 12 asks for a proof of this fact using mathematical induction).

Lemma 8.4. Let H be a subgraph of network $G = (V, E, c, s, t)$ that is induced by a subset $U \subseteq V - \{s, t\}$ of flow-conserving vertices. In other words, H is the subnetwork whose vertices are U and whose edges are all the edges $e \in E$ of the form $e = (u, v)$, where $u, v \in U$. Then, if f is a flow over G, then the amount of flow entering H equals the amount of flow leaving H.

Corollary 8.5. Given flow f over network $G = (V, E, c, s, t)$, the flow entering t equals $|f|$.

Proof of Proposition 8.2 Let f be a flow over $G = (V, E, c, s, t)$ and (X, Y) a cut of G. For $A, B \subseteq V$, $A \cap B = \emptyset$, we use the notation $f(A, B)$ to denote the sum of flow entering B along all edges $e = (a, b)$ for which $a \in A$ and $b \in B$.

Now given the subnetwork whose vertices are $Y - \{t\}$, by Lemma 8.4, the flow entering $Y - \{t\}$ must equal the flow leaving it, i.e.

$$f(X, Y - \{t\}) = f(Y, X) + f(Y, \{t\}),$$

which implies

$$f(X, Y - \{t\}) - f(Y, X) = f(Y, \{t\}).$$

Now, adding $f(X, \{t\})$ to both sides of the equation yields

$$f(X, \{t\}) + f(X, Y - \{t\}) - f(Y, X) = f(X, \{t\}) + f(Y, \{t\}),$$

which is equivalent (why?) to writing

$$f(X, Y) - f(Y, X) = |f|.$$

Finally, since $c(X, Y) \geq f(X, Y)$, we have

$$|f| = f(X, Y) - f(Y, X) \leq c(X, Y). \quad \square$$

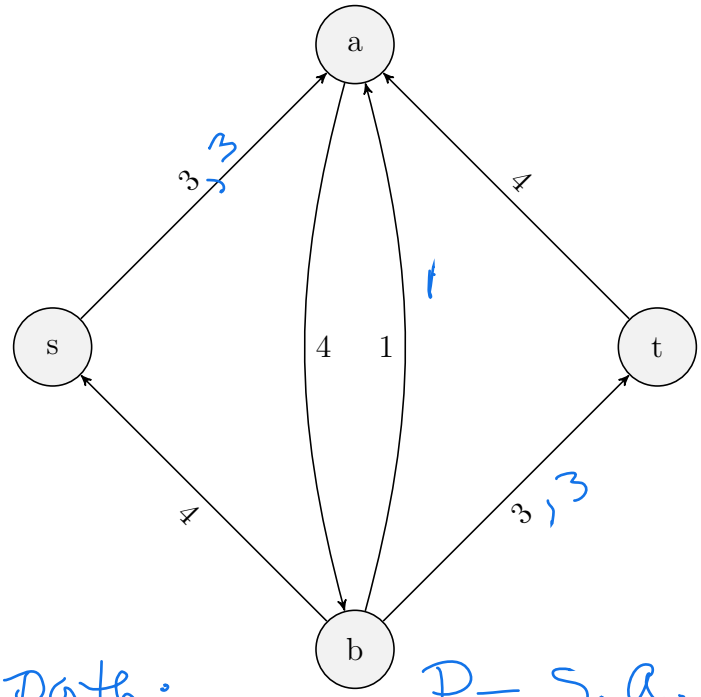
8.2 The Ford-Fulkerson Algorithm

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 8.6. The network below is the residual network G_f for the network G and flow f shown in Example 8.1.



Augmenting path: $P = s, a, b, t$

k = min capacity of edges (s,a), (a,b), and (b,t) =

$\min(3,4,3) = 3$

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 8.7. 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 8.7, 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 8.7 has the logical 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 by P. Then

$$0 \leq f'(e) = f(e) + \kappa \leq f(e) + (c(e) - f(e)) = c(e).$$

This is true since κ is bounded above by e's capacity in G_f which is $c(e) - f(e)$.

Case 2: e^r is traversed by P for some $e \in E$. Then

$$0 = f(e) - f(e) \leq f(e) - \kappa = f'(e) \leq f(e) \leq c(e).$$

This is true since the capacity of e^r is $f(e)$ and κ does not exceed this capacity.

Case 3: neither e nor e^r is traversed by P. Then the flow over e remains unchanged and so

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

since $f(e)$ already satisfies these inequalities because we are assuming that we are starting with a valid flow f.

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 the 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 $R \to Q$. Assume R: t is not reachable from s in G_f. Let $X \subseteq V$ be the set of all vertices that are reachable from s in G_f and let $Y = V - X$. Then (X, Y) is a cut of G_f. Moreover, it must be the case that there are no edges that start at X and end in Y. This is equivalent to saying that, in the original network G, given $x \in X$ and $y \in Y$, if $e = (x, y) \in E$, then $f(e) = c(e)$, and if $e = (y, x) \in E$, then $f(e) = 0$. In other words,

$$|f| = f(X, Y) - f(Y, X) = f(X, Y) - 0 = f(X, Y) = c(X, Y).$$

Therefore, f is a maximum flow since, by Proposition 8.2 no flow over G can exceed $c(X, Y)$. □

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 $e \in E$.

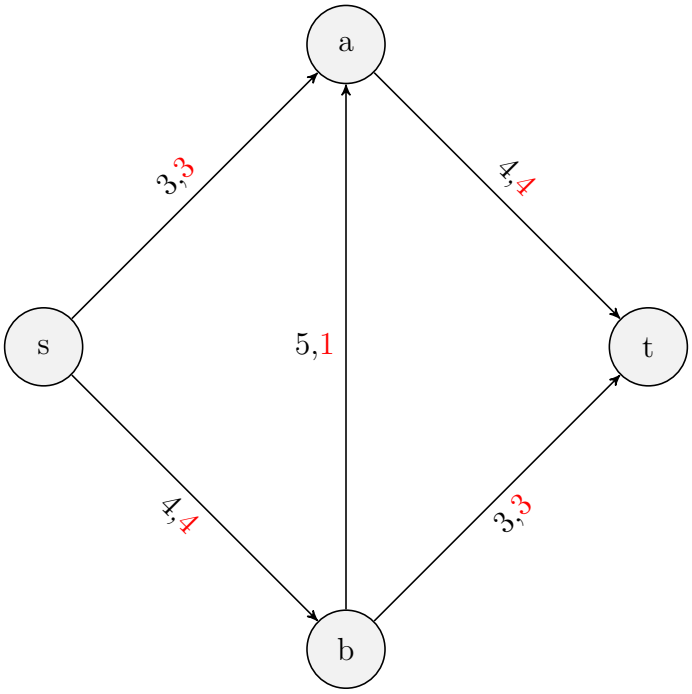
While `reachable`(G_f, s, t) is true,

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

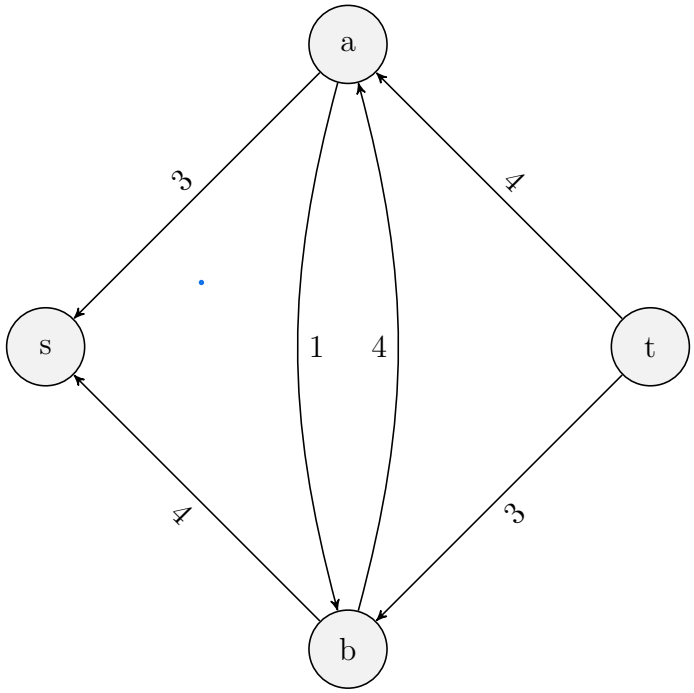
Return f.

Example 8.8. Starting with the flow f provided in Example 8.1, 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 8.6. 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.

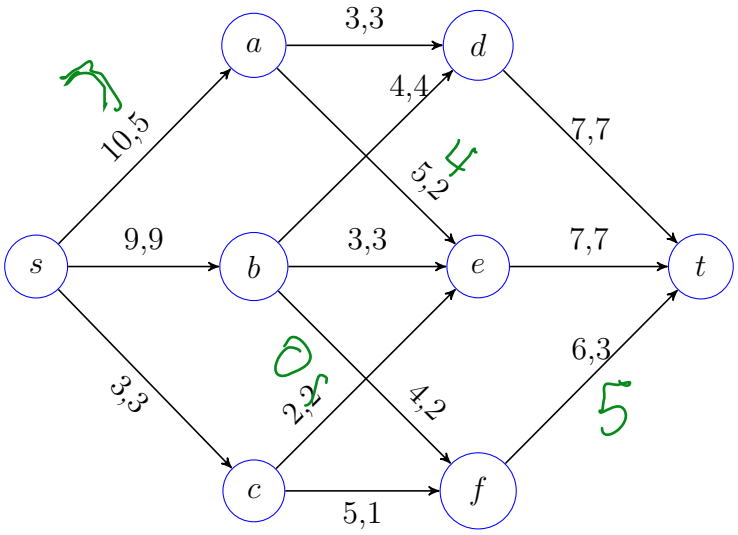
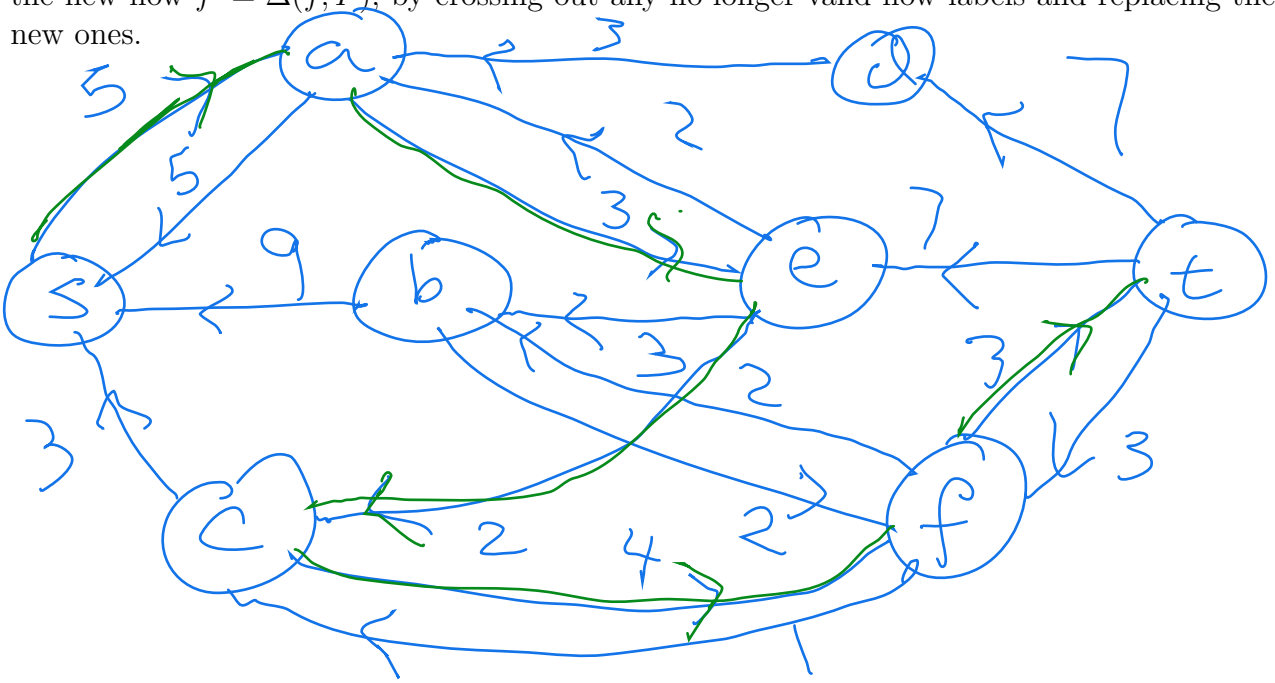


Figure 4: The network for Example 8.9.

Example 8.9. Consider the network $G = (V, E, c, s, t)$ shown in Figure 4, 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.



$P = s, a, e, c, f, t$

$k = \min(5, 3, 2, 4, 3) = 2$

Turing-Reducibility Exercises

1. Find a satisfying assignment for the set of clauses

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

2. Given `2SAT` instance $\mathcal{C}$, if $G_{\mathcal{C}}$ has path $P = x_2, \overline{x}_3, x_5, \overline{x}_1, \overline{x}_7$, then provide $\overline{P}$.

3. Given `2SAT` instance $\mathcal{C}$, if $G_{\mathcal{C}}$ has the inconsistent cycle $C = x_3, \overline{x}_4, x_5, x_1, x_4, \overline{x}_2, x_3$, then provide the subset $\mathcal{C}' \subseteq \mathcal{C}$ of clauses that is associated with this cycle. Which clauses in $\mathcal{C}'$ prevent a satisfying assignment from assigning $x_4 = 0$? Which clauses in $\mathcal{C}'$ prevent a satisfying assignment from assigning $x_4 = 1$? Conclude that $\mathcal{C}'$ (and $\mathcal{C}$ itself) is unsatisfiable.

4. 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.

5. 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.

6. Draw the implication graph for the following set of binary 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. Hint: remember that the first literal tested should be x_1, followed by $\overline{x}_1$ if necessary.

7. Repeat the previous problem, but now add the additional clause $(\overline{x}_2, x_3)$. Verify that there is now an inconsistent cycle in the implication graph by performing the `Improved 2SAT` algorithm and witnessing a variable x_i for which both R_{x_i} and $R_{\overline{x}_i}$ are inconsistent. Follow the hint from Exercise 6.

8. Draw the implication graph for the following instance of `2SAT`.

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

   Perform the `Improved 2SAT` algorithm to determine a satisfying assignment for this set of clauses. Follow the hint described in Exercise 6. Another hint: the final satisfying assignment should equal the union of two different consistent reachability sets.

9. 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?

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

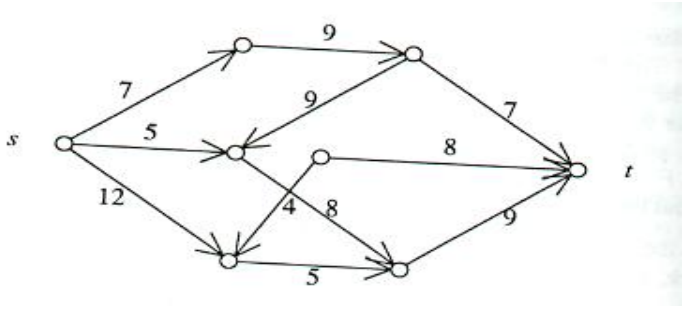
$$(s, a, 10), (s, b, 10), (a, c, 10), (b, d, 9), (b, e, 6), (c, b, 5), (c, t, 7), (d, e, 7), (d, t, 5), (e, t, 8).$$

    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.

11. 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.



12. Given a network $G = (V, E, c, s, t)$ and a positive flow f through G, consider 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. When adding an additional flow-conserving vertex to H, show that the in-flow and out-flow into H is still conserved.

13. Saffron Oil Company owns a collection of oil pump stations that are part of a larger network of stations. All pump stations are assumed to conserve oil, meaning that no oil is consumed by a station. Stations owned by Saffron are called **internal**, while those owned by other companies are called **external**. Moreover, since Saffron's stations lie in the interior of the grid, these stations only receive oil from other stations (either external or internal), and send oil to other stations (either external or internal). Currently, external stations pump oil to internal stations at a rate of $10,000$ gallons per hour. Likewise, internal stations pump oil to external stations at the same rate of $10,000$ gallons per hour. This rate G is referred to as the **company flow rate**. Saffron has just purchased an existing external station s. The flow rates (in gallons per hour) of oil entering and leaving s are i) 250 gallons per hour received from external stations, ii) 300 gallons per hour received from internal stations, iii) 400 gallons per hour sent to external stations, and iv) 150 gallons per hour sent to internal stations. Determine the new company flow rate after the purchase of s, and prove that the flow rate of oil received from external stations remains equal to the flow rate of oil sent to external stations.

Additional Exercises

A. Consider Marcia's algorithm from Example 3.1. Suppose the input numbers a and b both require n bits to represent. Explain why her algorithm requires at least 2^{n-1} steps. Conclude that Marcia's algorithm is *very* inefficient. Can you rewrite her program so that the number of steps gets reduced to $\mathrm{O}(n^2)$ steps and still makes use of an `Add`-oracle?

B. Suppose there is an algorithm that solves decision problem A. Why is it the case that $A \leq_{\mathrm{T}} B$ for any problem B?

C. 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.

D. Use the previous exercise as inspiration for describing an algorithm that establishes $A \leq_{\mathrm{T}} C$ in case both $A \leq_{\mathrm{T}} B$ and $B \leq_{\mathrm{T}} C$ are true.

E. Repeat the previous exercise, but replace $\leq_{\mathrm{T}}$ with $\leq_{\mathrm{T}}^{\mathrm{p}}$. In particular if $A \leq_{\mathrm{T}}^{p} B$ is established via an algorithm $\mathcal{A}_{AB}$ that requires $\mathrm{O}(n^k)$ steps and $B \leq_{\mathrm{T}}^{p} C$ is established via an algorithm $\mathcal{A}_{BC}$ that requires $\mathrm{O}(n^l)$ steps, then provide a worst-case number of steps that is required by the algorithm $\mathcal{A}_{AC}$.

F. 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.

G. 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.

Turing Reducibility Exercise Solutions

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

2. $P = x_7, x_1, \overline{x}_5, x_3, \overline{x}_2$.

3. Using the fact that $P \to Q$ is logically equivalent to $\overline{P} \vee Q$, we have

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

   If $x_4 = 0$, then clauses

$$(x_4, x_5), (\overline{x}_5, x_1), (\overline{x}_1, x_4)$$

   cannot all be satisifed. Indeed, if $x_4 = 0$, then the first clause forces $x_5 = 1$, which forces the second clause to assign $x_1 = 1$, which forces the third clause to assign $x_4 = 1$, which is a contradiction. Similarly, if $x_4 = 1$, then clauses

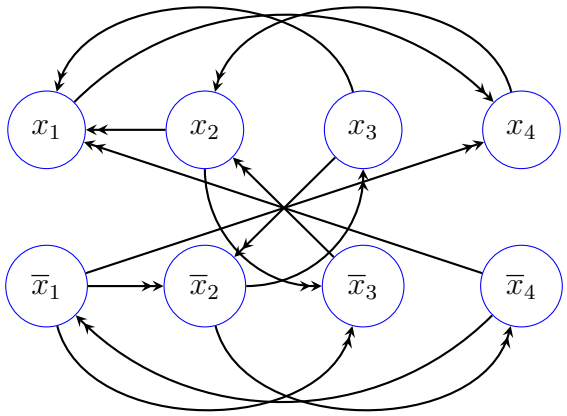
$$(\overline{x}_4, \overline{x}_2), (x_2, x_3), (\overline{x}_3, \overline{x}_4)$$

   cannot all be satisfied. Indeed, if $x_4 = 1$, then the first clause forces $x_2 = 0$, which forces the second clause to assign $x_3 = 1$, which forces the third clause to assign $x_4 = 0$, which is a contradiction. Thefore, any implication graph that contains such an inconsistent cycle as C leads to the original `2SAT` instance $\mathcal{C}$ being unsatisfiable.

4. 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.

5. 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"?

6. 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 $\mathcal{C}$



7. We have

$$R_{x_1} = R_{\overline{x_1}} = \{x_1, \overline{x_1}, x_2, \overline{x_2}, x_3, \overline{x_3}, x_4, \overline{x_4}\},$$

   are both inconsistent. Therefore, $\mathcal{C}$ is unsatisfiable.

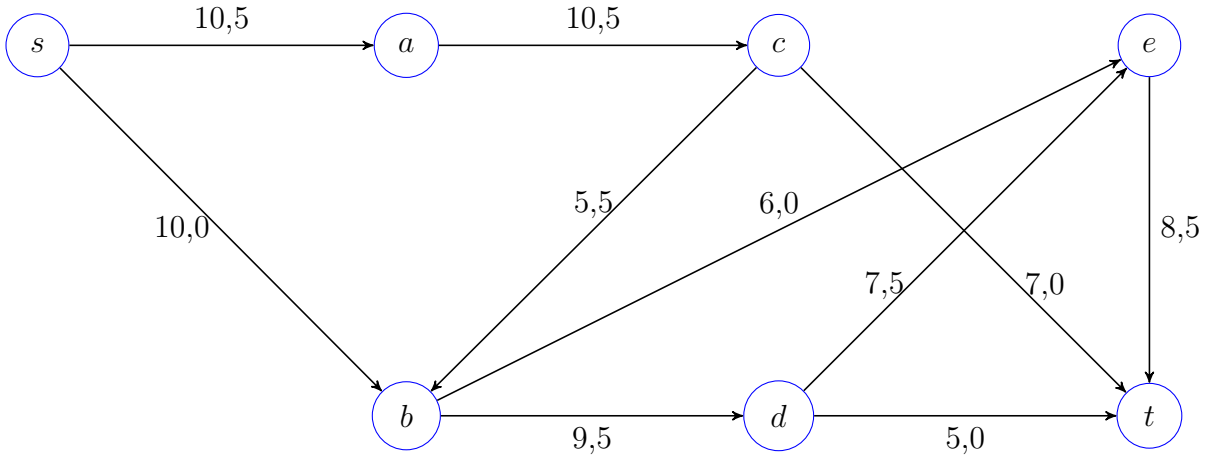


Figure 5: Network G with flow f_1

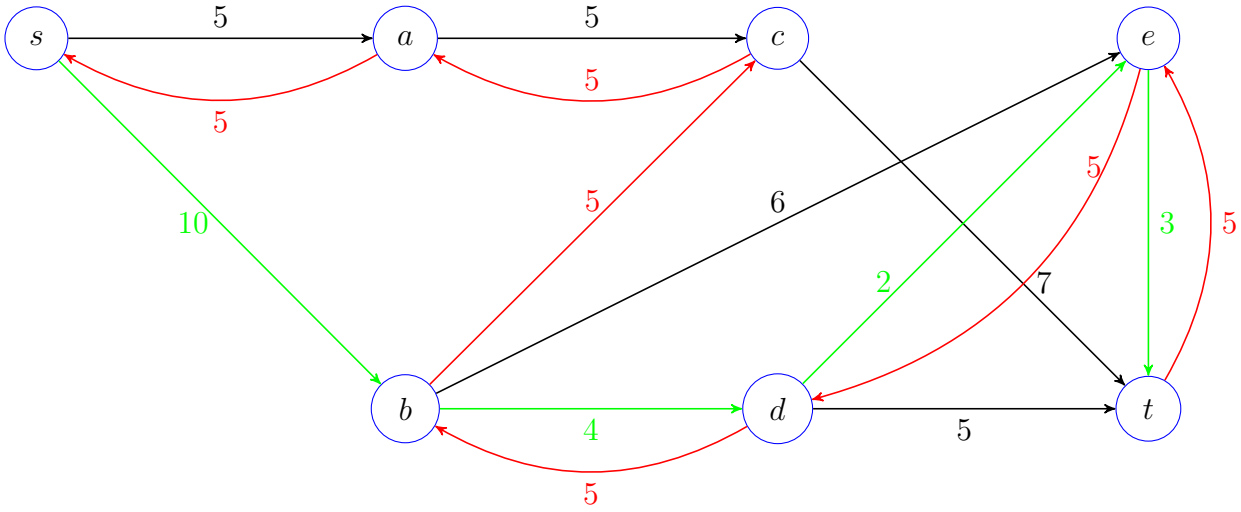


Figure 6: Residual Network G_{f_1} with P_1 in green and $\kappa = 2$

8. 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: $\alpha = (x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 1)$.

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

10. Maximum flow: $s(f) = 20$. See Figures 5 through 18. Note: in the residual networks, green edges are for the augmenting path, red for backward edges, and black for forward edges.

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

12. For proving the basis step, if H has a single vertex v, then, since v is flow-conserving and the

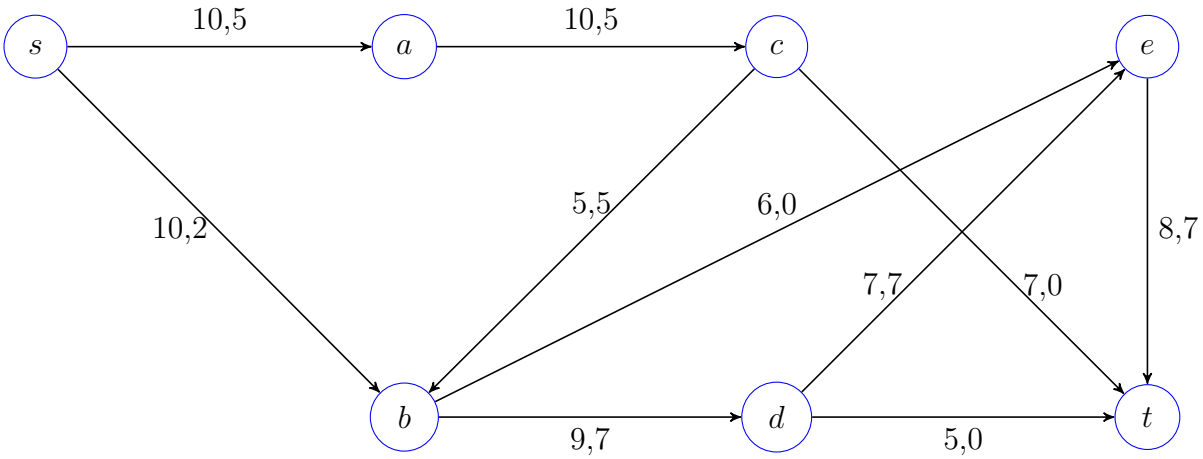


Figure 7: Network G with flow $f_2 = \Delta(f_1, P_1)$

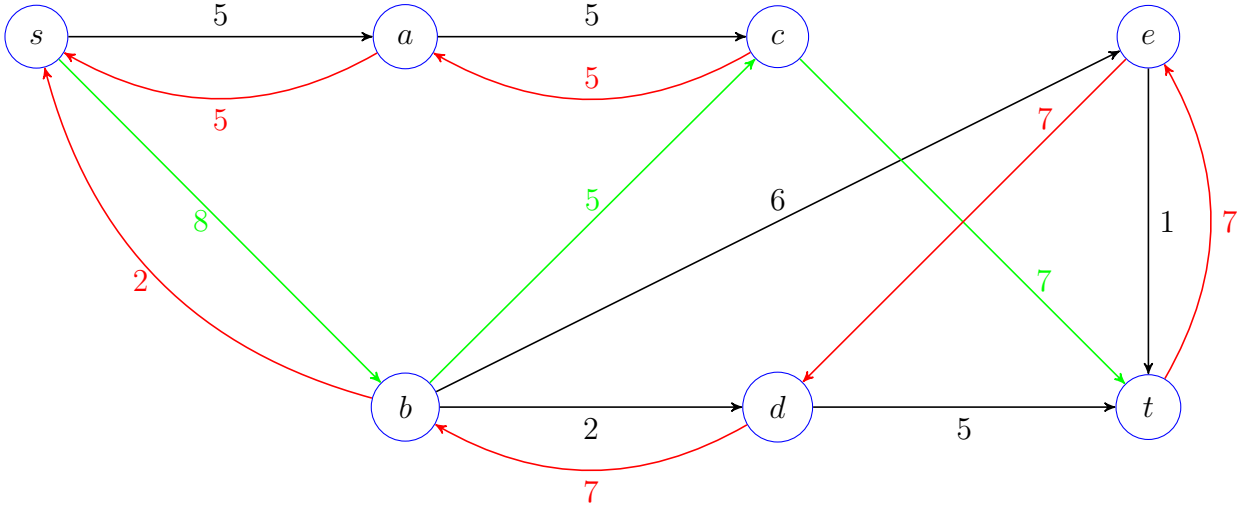


Figure 8: Residual Network G_{f_2} with P_2 in green and $\kappa = 5$

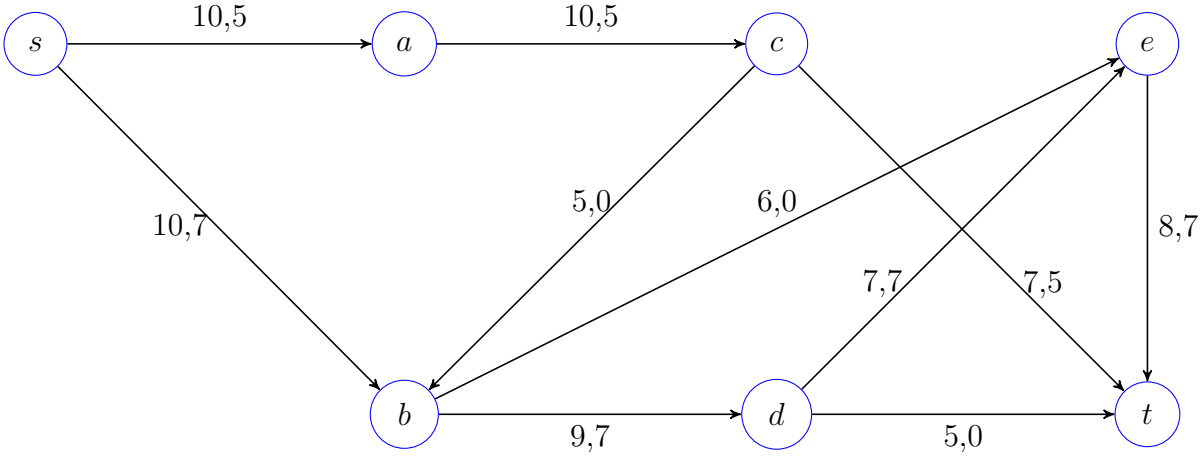


Figure 9: Network G with flow $f_3 = \Delta(f_2, P_2)$

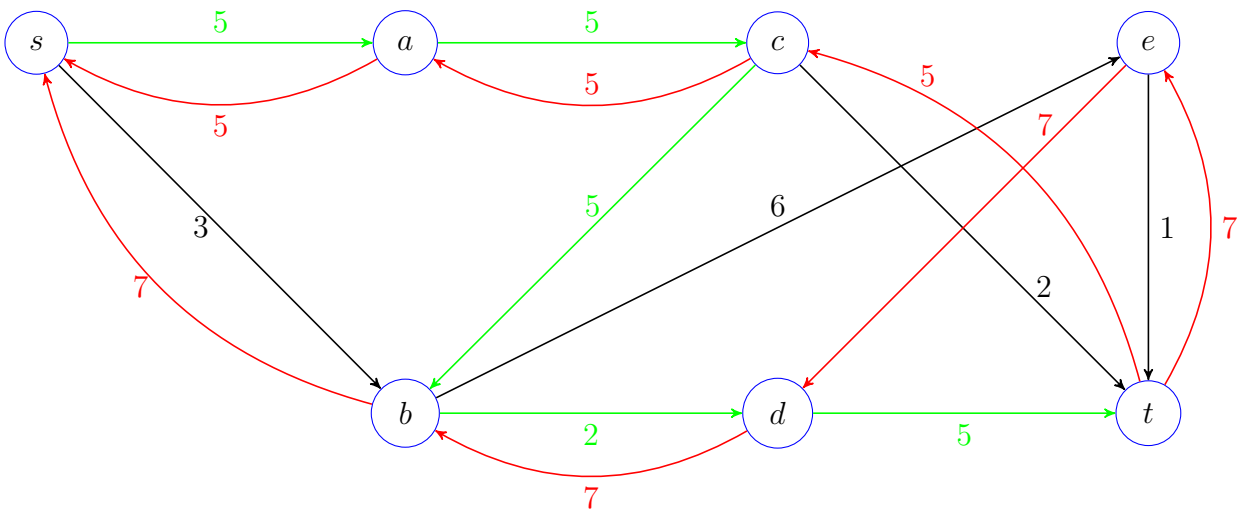


Figure 10: Residual Network G_{f_3} with P_3 in green and $\kappa = 2$. Note: this is an error since P_3 is NOT the longest path. I have not corrected it since it requires updating all subsequent figures)

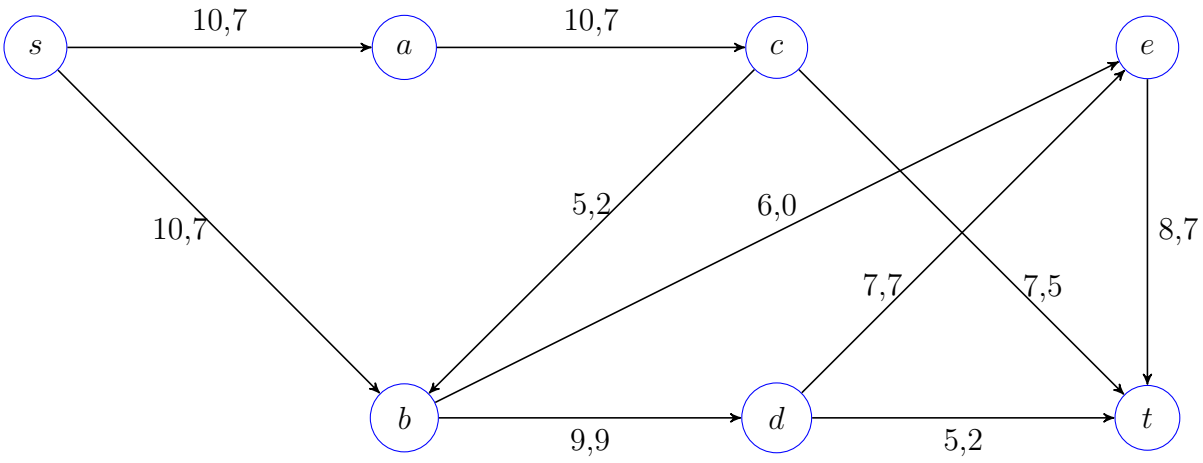


Figure 11: Network G with flow $f_4 = \Delta(f_3, P_3)$

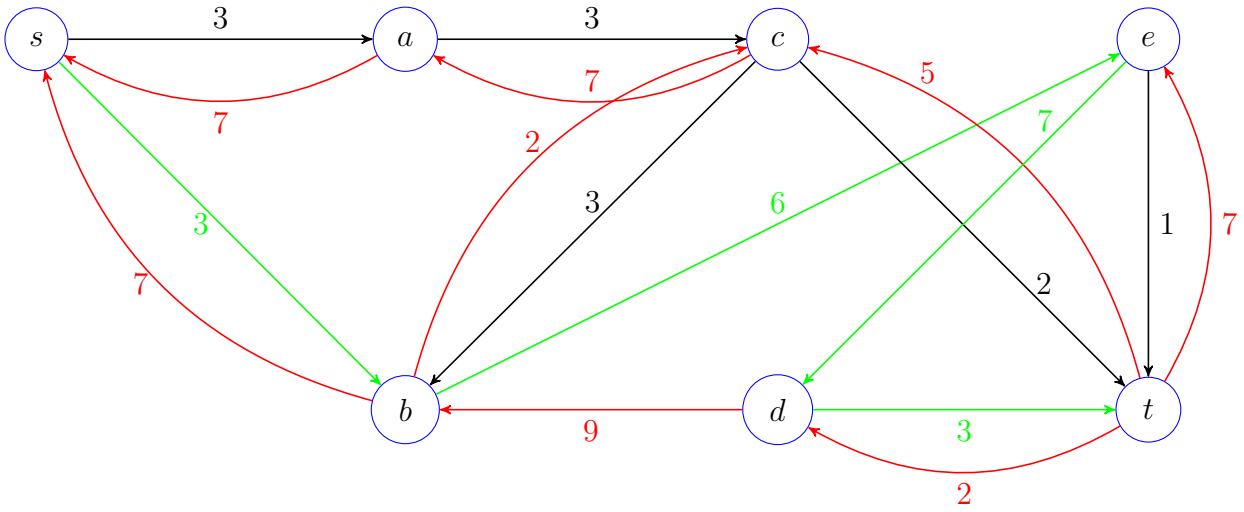


Figure 12: Residual Network G_{f_4} with P_4 in green and $\kappa = 3$ (Note: this is an error since P_4 is NOT the longest path. I have not corrected it since it requires updating all subsequent figures).

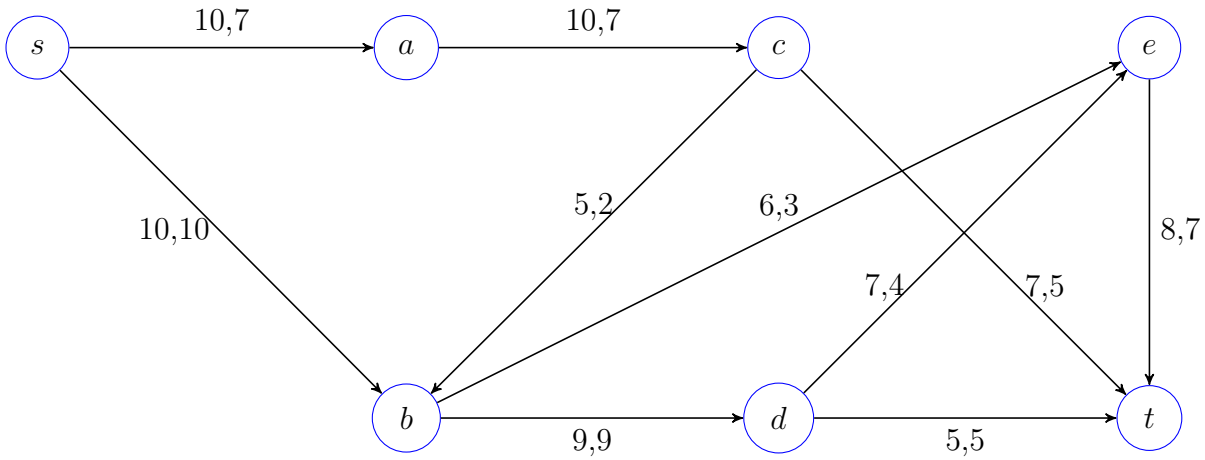


Figure 13: Network G with flow $f_5 = \Delta(f_4, P_4)$

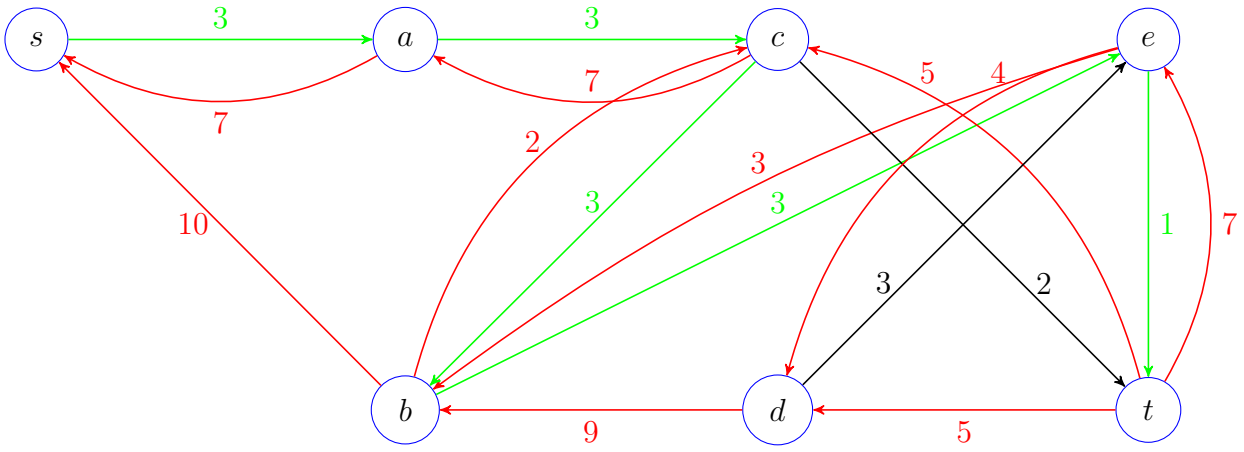


Figure 14: Residual Network G_{f_5} with P_5 in green and $\kappa = 1$

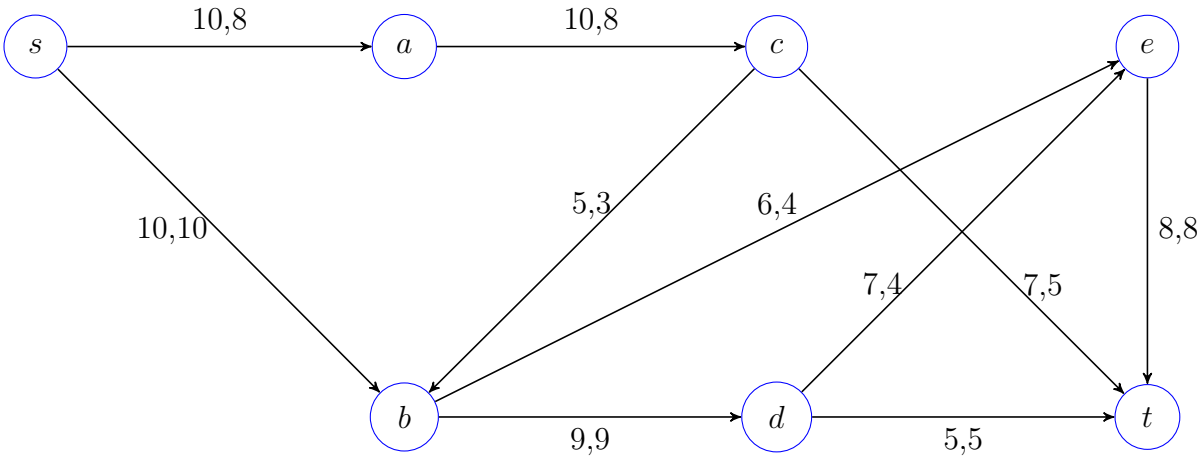


Figure 15: Network G with flow $f_6 = \Delta(f_5, P_5)$

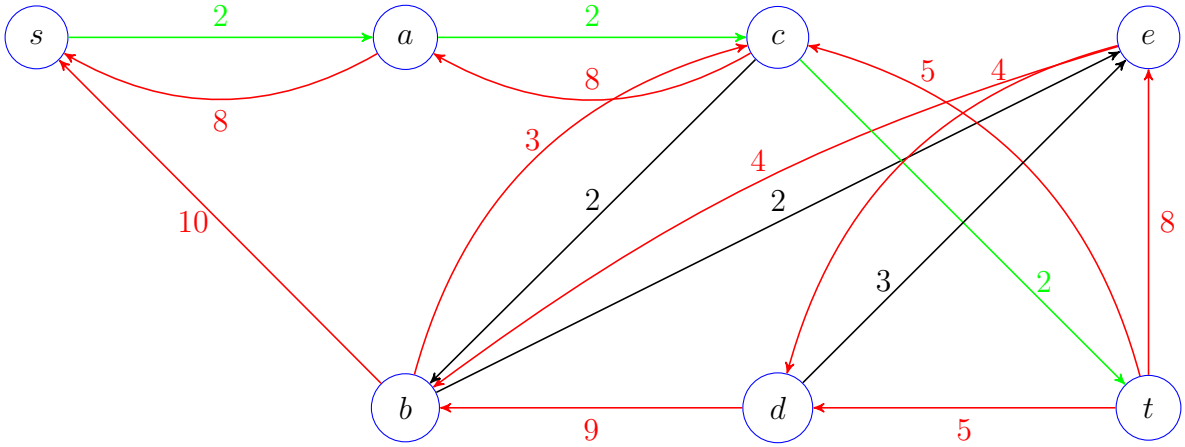


Figure 16: Residual Network G_{f_6} with P_6 in green and $\kappa = 2$

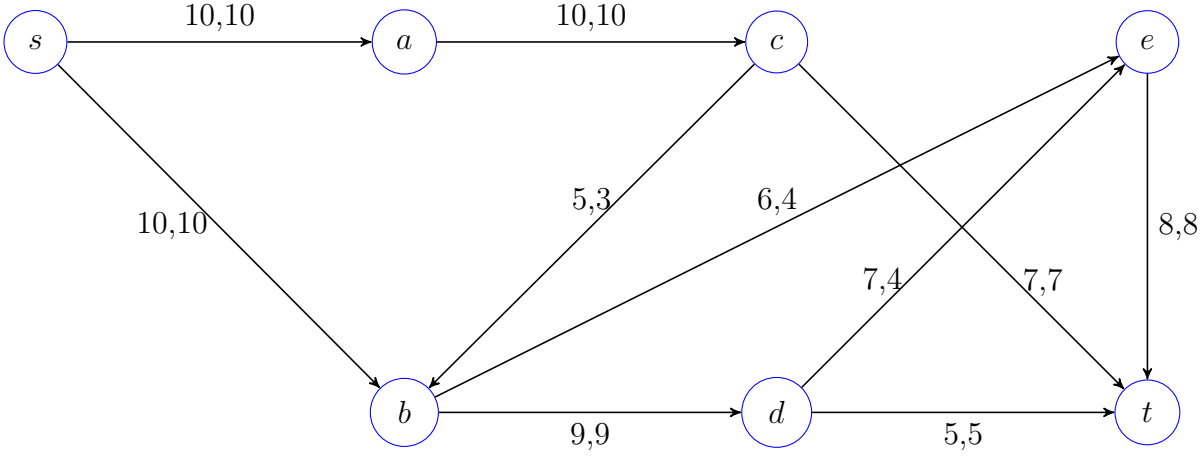


Figure 17: Network G with flow $f_7 = \Delta(f_6, P_6)$

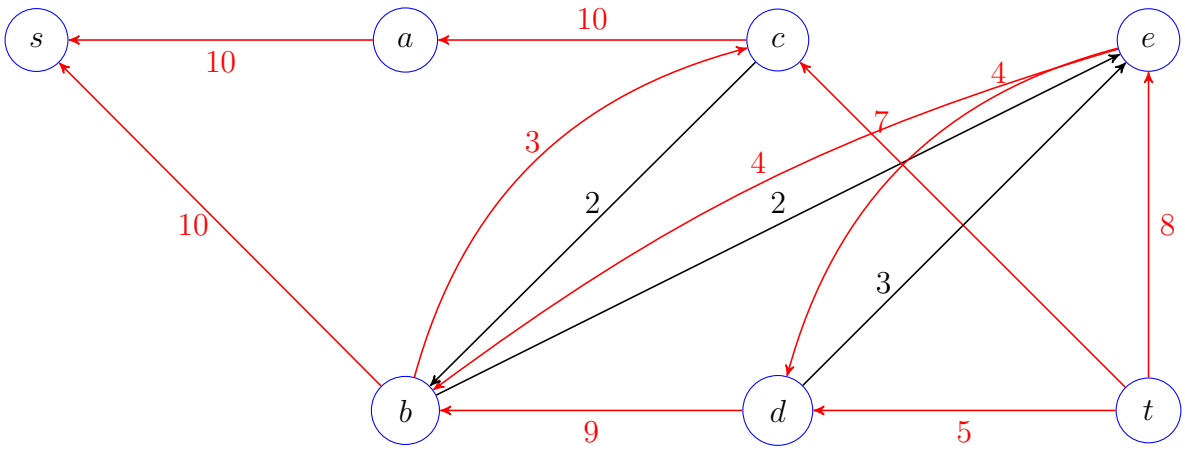


Figure 18: Residual Network G_{f_7} has no augmenting path from s to t. Algorithm terminates (it's about time!)

edges entering/leaving H are exactly the edges entering/leaving v, we have

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

For proving the inductive step, now suppose that any subgraph H consisting of $n-1$ flow-conserving vertices, $n \geq 2$, satisfies

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

where F is the total flow entering and leaving H. Let $\overline{H}$ denote all vertices of network G that are not in H.

Let $v \in \overline{H}$ be a flow-conserving vertex. Let H' denote the subnetwork induced by the vertices of H together with v. We show that the flow entering and leaving H' remains equal. Define the following values.

a. a denotes the sum of all flow entering v from H
b. b denotes the sum of all flow entering v from $\overline{H}$
c. c denotes the sum of all flow entering H from v
d. d denotes the sum of all flow entering $\overline{H}$ from v

Thus, since v is flow conserving, we have $a + b = c + d$. To prove the inductive step, we must show that an equal amount of flow enters and leaves H' which is comprised of n flow-conserving vertices. To this end, notice the following.

a. in-flow(H') = in-flow$(H) + b - c$. This is true since the flow entering v from $\overline{H}$ is now flow entering H', but the flow entering H from v is now internal flow for H'.
b. out-flow(H') = out-flow$(H) + d - a$. This is true since the flow leaving v to $\overline{H}$ is now flow leaving H', but the flow entering v from H is now internal flow for H'.

Therefore, we have established that the flow entering H' equals the flow leaving H' iff

$$b - c = d - a$$

which is true since $a + b = c + d$. This proves the inductive step and the result.

13. The new company flow rate is 10,100. See the proof of Exercise 12 for the rationale.

Solutions to Additional Exercises

A. Since the input numbers a and b both require n bits to represent, when written in binary, both numbers will have a 1 in the $(n-1)$ th bit place, and the value of both is at least 2^{n-1}. Therefore, the `for`-loop in the `multiply` program will iterate at least 2^{n-1} times.

B. Suppose A can be decided by some algorithm. Let B be any other decision 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 $A \leq_{\mathrm{T}} B$ by definition of Turing reducible, since the definition states that there exists an algorithm for solving A that makes "**zero** or more queries" to a B-oracle.

C. 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)));
}
```

D. 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.

E. 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, $A \leq_{\mathrm{T}}^{\mathrm{p}} C$.

F. **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).

G. 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.