Reductions and NP-Completeness

A giraffe with its long neck is a very different beast than a mouse, which is different from a snake. However, Darwin and gang observed that the first two have some key similarities, both being social, nursing their young, and having hair. The third is completely different in these ways. Studying similarities and differences between things can reveal subtle and deep understandings of their underlining nature that would not have been noticed by studying them one at a time. Sometimes things that at first appear to be completely different, when viewed in another way, turn out to be the same except for superficial cosmetic differences. This section will teach how to use reductions to discover these similarities between different optimization problems.

Reduction $P_1 \leq_{poly} P_2$: We say that we can *reduce* problem P_1 to problem P_2 if we can write a polynomial ($n^{\Theta(1)}$) time algorithm for P_1 using a supposed algorithm for P_2 as a subroutine. (Note we may or may not actually have an algorithm for P_2.) The standard notation for this is $P_1 \leq_{poly} P_2$.

Why Reduce? A reduction lets us compare the time complexities and underlying structures of these two problems. They are useful in providing algorithms for new problems (*upper bounds*), for giving evidence that there are no fast algorithms for certain problems (*lower bounds*), and for classifying problems according to their difficulty.

Upper Bounds: From the reduction $P_1 \leq_{poly} P_2$ alone, we cannot conclude that there is a polynomial time algorithm for P_1. But it does tell us that if there is a polynomial time algorithm for P_2, then there is one for P_1. This is useful in two ways. First, it allows us to construct algorithms for new problems from known algorithms for other problems. Moreover, it tells us that P_1 is "at least as easy as" P_2.

Hotdogs $\leq_{poly}$ Linear Programming: Section **??** describes how to solve the problem of making a cheap hotdog using an algorithm for solving linear programming.

Bipartite Matching $\leq_{poly}$ Network Flows: We will give an algorithm for Bipartite Matching in Section 0.4 that uses the network flows algorithm.

Lower Bounds: The contrapositive of the last statement is that if there is not a polynomial time algorithm for P_1, then there cannot be one for P_2 (otherwise there would be one for P_1.) This tells us that P_2 is "at least as hard as" P_1.

(Any Optimization Problem) $\leq_{poly}$ CIR-SAT: This small looking statement proved by Steve Cook in 1971 has become one of the foundations of theoretical computer science. There are many interesting optimization problems. Some people worked hard on discovering fast algorithms for this one and others did the same for that one. Cooks theorem

shows that it is sufficient to focus on the optimization problem CIR-SAT, because if you can solve it quickly then you can solve them all quickly. However, after many years of working hard, people have given up and highly suspect that at least one optimization problem that is hard. This gives strong evidence that CIR-SAT is hard. Cook's theorem is proved (and the problems defined) in Section 0.1.

CIR-SAT $\leq_{poly}$ 3-COL: See Section 0.3. This states that the optimization problem 3-COL is as hard as the already known to be hard problem CIR-SAT. This gives evidence that 3-COL is also hard. More over, reductions are transitive meaning that $P_1 \leq_{poly} P_2$ and $P_2 \leq_{poly} P_3$ automatically gives that $P_1 \leq_{poly} P_3$. Hence, together these last two statements give that (Any Optimization Problem) $\leq_{poly}$ 3-COL.

3-COL $\leq_{poly}$ Course Scheduling,

3-COL $\leq_{poly}$ Independent Set,

3-COL $\leq_{poly}$ 3-SAT:
These give evidence that Course Scheduling, Independent Set, and 3-SAT are hard. See Sections 0.2 and 0.3.

Halting Problem $\leq_{poly}$ (What Does This TM Do): It can be proved that the Halting Problem (given a Turing Machine M and an input I, does the M halt on I) is undecidable (no algorithm can always answer correctly in finite time). Given this, reductions can be used to prove that most any problem asking what the computation of a given Turing Machine does is also undecidable.

Reverse Reductions: Knowing $P_1 \leq_{poly} P_2$ and knowing that there is not a polynomial time algorithm for P_2 does not tell us anything about the whether there is a polynomial time algorithm for P_1. Though it does tell us that the algorithm for P_1 given in the reduction does not work, there well may be another completely different algorithm for P_1. Similarly, knowing that there is a polynomial time algorithm for P_1 does not tell us anything about the whether there one for P_2. To make these two conclusions, you must prove the reverse reduction $P_2 \leq_{poly} P_1$.

Classifying Problems: Reductions are used to classify problems.

The Same Problem except for Superficial Differences: More than just being able to compare their time complexities, knowing proving $P_1 \leq_{poly} P_2$ and $P_2 \leq_{poly} P_1$ reveals that the two problems are some how fundamentally at their core the same problem, asking the same types of questions. Sometimes this similarity is quite superficial. They simply use different vocabulary. However, at other times this connection between the problems is quite surprising, revealing a deeper understanding about each of the problems. One way in which we can make a reduction even more striking is by restricting the algorithm for the one to call the algorithm for the other only once. Then the mapping between them is even more direct.

NP-Complete: Above we showed that the optimization problems CIR-SAT, 3-COL, Course Scheduling, Independent Set, and 3-SAT, are all reducible to each other and in this way are all fundamentally the same problem. In fact there are thousands of very different problems that are equivalent to these. These problems are said to be *NP-Complete*. We discuss this more in section Section 0.2.

Halting Problem-Complete: Many classes of problems are defined in this way. Another important class consists of all problems that are equivalent in this way to the Halting Problem.

0.1 Satisfiability Is At Least As Hard As Any Optimization Problem

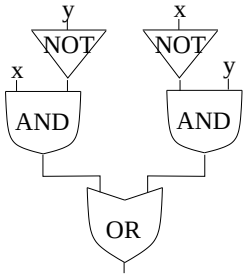
In Section **??** we saw that *optimization problems* involve searching through the exponential set of solutions for an instance to find one with optimal cost. Though there are quick algorithms (i.e. polynomial) for some of these problems, for most of them the best known algorithms require $2^{\Theta(n)}$ time on the worst case input instances and it is strongly believed that there are not polynomial time algorithms for them. The main reason for this belief is that many smart people have devoted many years of research looking for fast algorithms and have not found them. This section uses reductions to prove that some of these optimization problems are universally hard or *complete* amongst the class of optimization problems because if you could design an algorithm to solve such a problem quickly, then you could translate this algorithm into one that solves any optimization problem quickly. Conversely, (and more likely) if there is even one optimization problem that cannot be solved quickly, then none of these complete problems can be either. Proving in this way that a problem which your boss wants you to solve is hard is useful because you will know not to spend too much time trying to design an all purpose algorithm for it.

(Any Optimization Problem) $\leq_{poly}$ CIR-SAT: This reduction will prove the satisfiability problem is complete for the class of optimization problems, meaning that it is universally hard for this class.

The Circuit Satisfiability Problem: The famous computational problem, *circuit satisfiability* (*CIR-SAT*), is required to find a satisfying assignment for a given circuit. Section **??** gives a recursive backtracking algorithm for the satisfiability problem, but in the worst case its running time is $2^{\Theta(n)}$.

Circuit: A circuit is a both a useful notation for describing an algorithm in detail and a practical thing built in silicon in your computer.

Construction: It is built with *AND*, *OR*, and *NOT* gates. At the top are n wires labeled with the binary variables $x_1, x_2, \ldots, x_n$. To specify the circuit's input, each of these will take on either 1 or 0, *true* or *false*, 5 volts or 0 volts. Each *AND* gate has two wires coming into it, either from an input x_i or from the output of another gate. An *AND* gate outputs *true* if both of its inputs are *true*. Similarly, each *OR* gate outputs *true* if at least one of its inputs is *true* and each *NOT* gate outputs *true* if its single input is *false*. We will only consider circuits that have no cycles, so these *true/false* values percolate down to the output wires. There will be a single output wire if the circuit computes a *true/false* function of its input $x_1, x_2, \ldots, x_n$ and will have m output wires if it outputs an m bit string which can be used to encode some required information.



A circuit for $x \oplus y$

Compute Any Function: Given any function $f : \{0,1\}^n \to \{0,1\}^m$, a circuit can compute it with a most $\Theta(nm \cdot 2^n)$ gates as follows. For any fixed input instance $\langle x_1, x_2, \ldots, x_n \rangle = \langle 1, 0, \ldots, 1 \rangle$, a circuit can say "the input is this fixed instance" simply by computing $[(x_1 = 1)\ AND\ NOT(x_2 = 1)\ AND\ \ldots\ AND\ (x_n = 1)]$. Then the circuit computes the i^{th} bit of the function's output by outputting 1 if ["the input is this fixed instance" OR "this instance" $OR \ldots OR$ "this instance"], where each instance is listed for which the i^{th} bit of the function's output is 1.

Poly Size for Poly Time: More importantly, given any algorithm whose Turing Machine's running time is $T(n)$ and given any fixed integer n, there is an easily constructed circuit with at most $\Theta(T(n)^2)$ gates that computes the output of the algorithm given any n bit input instance. Change the definition of a Turing machine slightly so that each cell is big enough so that the cell currently being pointed to by the head can store not only its current cell's contents but also the current state of the machine. This cell's contents can be encoded with $\Theta(1)$ bits. Because the Turing machine uses only $T(n)$ time, it can use at most the first $T(n)$ cells of memory. For each of the $T(n)$ steps of the algorithm, the circuit will have a row of $\Theta(1) \cdot T(n)$ wires whose values encode the contents of memory of these $T(n)$ cells during this time step. The gates of the circuit between these rows of wires compute the next contents of memory from the current contents. Because the contents of cell i at time t depends only on the contents of cells $i-1$, i, and $i+1$ at time $t-1$ and each of these is only a $\Theta(1)$ number of bits, this dependency can computed using a circuit with $\Theta(1)$ gates. This is repeated in a matrix of $T(n)$ time steps and $T(n)$ cells for a total of $\Theta(T(n)^2)$ gates. At the bottom, the circuit computes the output of the function from the contents of memory of the Turing machine at time $T(n)$.

Circuit Satisfiability Specification: The CIR-SAT problem take as input a circuit with a single *true/false* output and return an assignment to the variables $x_1, x_2, \ldots, x_n$ for which the circuit gives *true*, if such an assignment exists.

Optimization Problems: This reduction will select a generic optimization problem and show that CIR-SAT is at least as hard as it is. To do this, we need to have a clear definition of what a generic optimization problem looks like.

Definition: Each such problem has a set instances that might be given as input, each instance has a set of potential solutions some of which are valid, and each solution has a cost. The goal, given an instance, is to find one of its valid solutions with optimal cost. An important feature of an optimization problem is that there are polynomial time algorithms for the following.

***Valid*(I, S):** Given an instance I and a potential solution S, there is an algorithm $Valid(I, S)$ running in time $|I|^{\mathcal{O}(1)}$ that determines if I is a valid instance for the optimization problem and that S is a valid solution for I.

***Cost*(S):** Given a valid solution S, there is an algorithm $Cost(S)$ running in time $|I|^{\mathcal{O}(1)}$ that computes the cost of the solution S.

Example: Course Scheduling Given the set of courses requested by each student and the set of time slots available, find a schedule which minimizes the number of conflicts.

***I*:** The set of courses requested by each student and the set of time slots available

***S*:** A schedule

***Valid*(I, S):** An algorithm which returns whether the schedule S is valid for the courses and student requests given in I.

***Cost*(S):** An algorithm which returns the number of conflicts in the schedule S.

***Alg*$_{alg}$ for the Optimization Problem:** Given a fast algorithm Alg_{oracle} for CIR-SAT and the descriptions $Valid(I, S)$ and $Cost(S)$ of an optimization problem, we will now design a fast algorithm Alg_{alg} for the optimization problem and use it to prove that the problem CIR-SAT is "at least as hard as" the optimization problem

Binary Search for Cost: Given some instance I_{alg} to the optimization problem, Alg_{alg}'s first task is to determine the cost c_{opt} of the optimal solution for I. Alg_{alg} starts by determining whether or not there is a valid solution for I that has cost at least $c = 1$. If it does, Alg_{alg} repeats this with $c = 2, 4, 8, 16, \ldots$. If it does not, Alg_{alg} tries $c = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots$, until it finds c_1 and c_2 between which it knows the cost of an optimal solution lies. Then it does binary search to find c_{opt}. The last step is to find a solution for I that has this optimal cost.

Finding a Solution with given Cost: Alg_{alg} determines whether I has a solution S with cost at least c or finds a solution with cost c_{opt} as follows. Alg_{alg} constructs a circuit C and calls the algorithm Alg_{oracle}, which provides a satisfying assignment to C. Alg_{alg} wants the satisfying assignment that Alg_{oracle} provides to be the solution S that it needs. Hence, Alg_{alg} designs C to be satisfied by the assignment S only if S is a solution S for I with a cost as required, i.e., $C(S) \equiv [Valid(I, S)$ and $Cost(S) \geq c]$. Because there are polynomial time algorithms for $Valid(I, S)$, for $Cost(S)$, and for $\geq$, Alg_{alg} can easily construct such a circuit $C(S)$. If a such solution S satisfying C exists, then Alg_{oracle} kindly provides one.

This completes the reduction of any Optimization problem to CIR-SAT.

Exercise 0.1.1 *For each of the following problems, define I, S, $Valid(I, S)$ and $Cost(S)$.*

1. **Graph Colouring:** *Given a graph, colour the nodes of the graph so that two nodes do not have the same colour if they have an edge between them. Use as few colours as possible.*

2. **Independent Set:** *Given a graph, find a largest subset of the nodes for which there are no edges between any pair in the set.*

3. **Airplane:** *Given the requirements of a plane, design it optimizing its performance.*

4. **Business:** *Given a description of the business, make a business plan to maximize its profits.*

5. **Factoring:** *Given an integer, factor it, eg.* $6 = 2 \times 3$.

6. **Cryptography:** *Given an encrypted message, decode it.*

0.2 Steps to Prove NP-Completeness

In this section we define the class NP and give steps for proving that a computational problem is NP-complete.

Complete for Non-Deterministic Polynomial Time Decision Problems: The set of computational problems that are complete (universally hard) for optimization problems is extremely rich and varied. Studying them has become a fascinating field of research.

NP Decision Problems: Theoretical computer scientists generally only consider a subclass of the optimization problems referred to as the class of *Non-Deterministic Polynomial Time* Problems (NP).

One Level of Cost: Instead of worrying about whether one solution has a better cost than another, we will completely drop the notion of the cost of a solution. S will only be considered to be a "valid solution" for the instance I if it is a solution with a sufficiently good cost. This is not a big restriction, because if you want to consider solutions with different costs, you can always do binary search as done above for the cost of the optimal solution.

Witness: A solution for an instance is often referred to as a *witness*, because though it may take exponential time to find it, if it were provided by a (non-deterministic) fairy god mother, then it can be used in polynomial time to witness the fact that the answer for this instance is yes. In this way, NP problems are asymmetrical because there does not seem to be a witness that quickly proves that an instances does not have solution.

Formal Definition: We say that such a computational problem P is in the class of *Non-Deterministic Polynomial Time* Problems (NP) if there is a polynomial time algorithm $Valid(I, S)$ that specifies *Yes* when S is a (sufficiently good) solution for the instance I and *No*, if not. More formally, P can be defined as follows.

$$P(I) \equiv [\exists S,\ Valid(I, S)]$$

Examples:

Circuit Satisfiability (CIR-SAT): Circuit Satisfiability was initially defined as a decision problem. Given a circuit, determine whether there is an assignment that satisfies it.

Graph 3-Colouring (3-COL): Given a graph, determine whether its nodes can be coloured with three colours so that two nodes do not have the same colour if they have an edge between them.

Course Scheduling: Given the set of courses requested by each student, the set of time slots available, and an integer K, determine whether there is schedule with at most K conflicts.

Cook vs Karp Reductions: Stephen Cook first proved that CIR-SAT is complete for the class of NP-problems. His definition of a reduction $P_{alg} \leq_{poly} P_{oracle}$ is that one can write an algorithm Alg_{alg} for the problem P_{alg} using an algorithm Alg_{oracle} for the problem P_{oracle} as a subroutine. In general, this algorithm Alg_{alg} could call Alg_{oracle} as many times as it likes and do anything it wants with the answers that it receives. Richard Karp later observed that when the problems P_{alg} and P_{oracle} are more similar in nature then the algorithm Alg_{alg} used in the reduction need only call Alg_{oracle} once and answers *Yes* if and only if Alg_{oracle} answers *Yes*. These two definitions of reductions are referred to as *Cook* and *Karp* reductions. Though we defined Cook reductions above because they are more natural, we will consider only Karp reductions from here on.

NP-Complete: We say that a computational problem P_{oracle} is *NP-complete* if

1. it is in NP and
2. every language in NP can be polynomially reduced to it using a Karp reduction. More formally,

$$\forall \text{ Optimization Problems } P_{alg},\ P_{alg} \leq_{poly} P_{oracle}.$$

To prove this, it is sufficient to prove that that our computational problem is at least as hard as some problem already known to be NP-complete. For example, because we now know that CIR-SAT is NP-complete, it is sufficient to prove that CIR-SAT $\leq_{poly} P_{oracle}$.

The Steps to Prove NP-Complete: Proving problems to be NP-complete is a bit of an art, but once you get the hang of it, they can be quite fun problems to solve. We will now carefully lay out the steps needed.

Running Example: Course Scheduling is NP-Complete: We will use the steps to prove that Alg_{oracle} = Course Scheduling problem is NP-complete.

0) $P_{oracle} \in$ NP: The first step is to prove that problem P_{oracle} is in NP by providing the polynomial time algorithm $Valid(I_{oracle}, S_{oracle})$ that specifies whether S_{oracle} is a valid solution for instance I_{oracle}.

Course Scheduling: It is not hard to determine in polynomial time whether the instance I_{oracle} and the solution S_{oracle} are properly defined and to check that within this schedule S_{oracle}, the number of times that a student wants to take two courses that are offered at the same time is at most K.

1) What To Reduce To It: An important and challenging step in proving that a problem is NP-complete is deciding which NP-complete problem to reduce to it.

3-COL $\leq_{poly}$ Course Scheduling: We will reduce 3-COL to Course Scheduling, that is, we will prove the reduction 3-COL $\leq_{poly}$ Course Scheduling. We will save the proof that 3-COL is NP-complete for our next example because it is much harder.

Hint: You want to choose a problem that is "similar" in nature to yours. In order to have more to choose from, it helps to know a large collection of problems that are NP-complete. There are entire books devoted to this. When in doubt 3-SAT and 3-Col are good problems to use.

2) What is What: It is important to remember what everything is.

3-COL $\leq_{poly}$ Course Scheduling:

P_{alg} = 3COL is the Graph 3-Colouring problem.

$I_{alg} = I_{graph}$, an instance to it, is an undirected graph.

$S_{alg} = S_{colouring}$, a potential solution, is a colouring of each of its nodes with either red, blue or green. It is a valid solution if no edge has two nodes with the same colour.

$\boldsymbol{Alg_{alg}}$ is an algorithm that takes graph I_{graph} as input and determines whether it has a valid colouring.

$\boldsymbol{P_{oracle}}$ **= Course Scheduling**

$\boldsymbol{I_{oracle} = I_{courses}}$**,** an instance to it, is the set of courses requested by each student, the set of time slots available, and the integer K.

$\boldsymbol{S_{oracle} = S_{schedule}}$**,** a potential solution, is a schedule assigning courses to time slots. It is a valid solution if it has at most K conflicts.

$\boldsymbol{Alg_{oracle}}$ is an algorithm that takes $I_{courses}$ as input and determines whether it has a valid schedule.

Such instances which may or may not be satisfiable and such potential solutions may or may not be valid until they are proved.

Warning: Be especially careful about what is an instance and what is a solution for each of the two problems.

3) Direction of Reduction and Code: Another common source of mistakes is doing the reduction in the wrong direction. I recommend not memorizing this direction, but working it out each time. Our goal is to prove that the problem P_{oracle} is at least as hard as P_{alg}, namely $P_{alg} \leq_{poly} P_{oracle}$. We prove that P_{alg} is relatively easy by designing a fast algorithm Alg_{alg} for it using a supposed fast algorithm Alg_{oracle} for P_{oracle}. (But there is likely not a fast algorithm for P_{alg} and hence there is not likely one for P_{oracle}.) The code for our algorithm for P_{alg} will be as follows.

algorithm $Alg_{alg}(I_{alg})$

$\boldsymbol{\langle pre{-}cond \rangle}$**:** I_{alg} is an instance of P_{alg}.

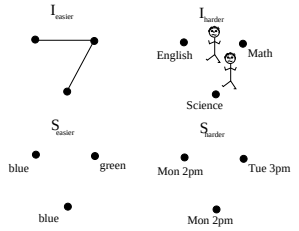
$\boldsymbol{\langle post{-}cond \rangle}$**:** Determine whether I_{alg} has a solution S_{alg} and if so returns it.

begin
    $I_{oracle} = InstanceMap(I_{alg})$
    $\langle ans_{oracle}, S_{oracle} \rangle = Alg_{oracle}(I_{oracle})$
    if($ans_{oracle} = Yes$) then
        $ans_{alg} = Yes$
        $S_{alg} = SolutionMap(S_{oracle})$
    else
        $ans_{alg} = No$
        $S_{alg} = nil$
    end if
    return($\langle ans_{alg}, S_{alg} \rangle$)
end algorithm

4) Look For Similarities: Though the problems P_{alg} and P_{oracle} may appear to be very different, the goal in this step is to look for underlying similarities. Compare how their solutions, S_{alg} and S_{oracle}, are formed out of their instances, I_{alg} and I_{oracle}. Generally, the instances can be thought of as sets of elements with a set of constraints between them. Can you view their solutions as subsets of these elements or as a labeling of them? What allows a solution to form and what constrains how it is formed. Can you talk about the two problems using the

same language? For example, a subset of elements can be viewed as a labeling of the elements with zero and one. Similarly, a labeling of each element e with $\ell_e \in [1..L]$ can be viewed as a subset of the pairs $\langle e, \ell_e \rangle$.

3-COL $\leq_{poly}$ Course Scheduling: A solution $S_{colouring}$ is a colouring that assigns a colour to each node. A solution $S_{schedule}$ is a schedule that assigns a time slot to each course. This similarity makes it clear that there is a similarity between the roles of the nodes of I_{graph} and of the courses of $I_{courses}$ and between the colours of $S_{colouring}$ and the time slots of $S_{schedule}$. Each colouring conflict arises from an edge between nodes and each scheduling conflict arises from a student wanting two courses. This similarity makes it clear that there is a similarity between the roles of the edges of I_{graph} and of the course requests of $I_{courses}$.



5) InstanceMap: You must define a polynomial time algorithm $InstanceMap(I_{alg})$ that, given an instance I_{alg} of P_{alg}, constructs an instance I_{oracle} of P_{oracle} that has "similar" sorts of solutions. The main issue is that the constructed instance I_{oracle} has a solution if and only if the given instance I_{alg} has a solution, that is *Yes* instances get mapped to *Yes* instances and *No* to *No*.

3-COL $\leq_{poly}$ Course Scheduling: Given a graph I_{graph} to be coloured, we design an instance $I_{courses} = InstanceMap(I_{graph})$ to be scheduled. Using the similarities observed in step 4, our mapping we will have one course for each node of the graph and one time slot for each of the three colours green, red, and blue. For each edge between nodes u and v in the graph, we will have a student who requests both course u and course v. The colouring problem does not allow any conflicts. Hence, we set $K = 0$.

Not Onto or 1-1: It is important that each instance I_{alg} is mapped to some instance I_{oracle}, but it is not important whether an instance P_{oracle} is mapped to more than one or none at all. In our example, we never mention instances to be scheduled that have more than three time slots or that allow $K > 0$ conflicts.

Warning: Be sure to do this mapping in the correct direction. The first step in designing an algorithm Alg_{alg} is to suppose that you have been given an input I_{alg} for it. Before your algorithm can call the algorithm Alg_{oracle} as a subroutine, your must construct an instance I_{oracle} to give to it.

Warning: Do not define the mapping only for the *Yes* instances or use a solution S_{alg} for I_{alg} for determining the instance I_{oracle} mapped to. The algorithm Alg_{alg} that you are designing is given an instance I_{alg}, but it does not know whether or not the instance has a solution. The whole point is to give an argument that finding a solution may take exponential time. It is safer when defining the mapping $InstanceMap(I_{alg})$ not to even mention whether the instance I_{alg} has a solution or what that solution might be.

6) SolutionMap: You must also define a polynomial time algorithm $SolutionMap(S_{oracle})$ mapping each valid solution S_{oracle} for the instance $I_{oracle} = InstanceMap(I_{alg})$ you just constructed to a valid solution S_{alg} for instance I_{alg} that was given as input. Valid solutions can be subtle and the instance I_{oracle} may have some solutions that you had not intended

when you constructed it. One way to help avoid missing some is to throw a much wider net by considering all "potential solutions." In this step, for each potential solution S_{oracle} for I_{oracle}, you must either give a reason why it is not a valid solution or map it to a solution $S_{alg} = SolutionMap(S_{oracle})$ for I_{alg}. It is fine if some of the solutions that you map happen not to be valid.

3-COL $\leq_{poly}$ Course Scheduling: Given a schedule $S_{schedule}$ assigning course u to time slots c, we define $S_{colouring} = SolutionMap(S_{schedule})$ to be the colouring that colours node u with colour c.

Warning: When the instance I_{oracle} you constructed has solutions that you did not expect, there are two problems. First, the unknown algorithm Alg_{oracle} may give you one of these unexpected solutions. Second, there is a danger that I_{oracle} has solutions but your given instance I_{alg} does not. For example, if in step 5, our I_{oracle} allowed more than three time slots or more than $K = 0$ conflicts, then the instance may have many unexpected solutions. In such cases, you may have to redo step 5 adding extra constraints to the instance I_{oracle} so that it no longer has these solutions.

7) Valid to Valid: In order to prove that the algorithm $Alg_{alg}(I_{alg})$ works, you must prove that if S_{oracle} is a valid solution for $I_{oracle} = InstanceMap(I_{alg})$, then $S_{alg} = SolutionMap(S_{oracle})$ is a valid solution for I_{alg}.

3-COL $\leq_{poly}$ Course Scheduling: Supposing that the schedule is valid, we prove that the colouring is valid as follows. The instance to be scheduled is constructed so that, for each edge of the given graph, there is a student who requests the courses u and v associated with the nodes of this edge. Because the schedule is valid, there are $K = 0$ course conflicts, and hence these courses are all scheduled at different time slots. The constructed colouring, therefore allocates different colours to these nodes.

8) ReverseSolutionMap: Though we do not need it for the code, for the proof you must define an algorithm $ReverseSolutionMap(S'_{alg})$ mapping in the reverse direction from each "potential" solution S'_{alg} for instance I_{alg} to a potential solution S'_{oracle} for instance I_{oracle}.

3-COL $\leq_{poly}$ Course Scheduling: Given a colouring $S'_{colouring}$ colouring node u with colour c, we define $S'_{schedule} = ReverseSolutionMap(S'_{colouring})$ to be the schedule assigning course u to time slots c.

Warning: $ReverseSolutionMap(S'_{alg})$ does not need to be the inverse map of $SolutionMap(S_{oracle})$. You must define the mapping $ReverseSolutionMap(S_{alg})$ for every possible solution S'_{alg}, not just those mapped to by $SolutionMap(S_{oracle})$. Otherwise, there is the danger is that I_{alg} has solutions but your constructed instance I_{oracle} does not.

9) Reverse Valid to Valid: You must also prove the reverse direction that if S'_{alg} is a valid solution for I_{alg}, then $S'_{oracle} = ReverseSolutionMap(S'_{alg})$ is a valid solution for $I_{oracle} = InstanceMap(I_{alg})$.

3-COL $\leq_{poly}$ Course Scheduling: Supposing that the colouring is valid, we prove that the schedule is valid as follows. The instance to be scheduled is constructed so that each student requests the courses u and v associated with nodes of some edge. Because

the colouring is valid, these nodes have been allocated different colours and hence the courses are all scheduled different time slots. Hence, there will be $K = 0$ course conflicts.

10) Working Algorithm: Given the above steps, it is now possible to prove that if the supposed algorithm Alg_{oracle} correctly solves P_{oracle}, then our algorithm Alg_{alg} correctly solves P_{alg}.

Yes to Yes: We start by proving that Alg_{alg} answers *Yes* when given an instance for which the answer is *Yes*. If I_{alg} is a *Yes* instance, then by the definition of the problem P_{alg}, it must have a valid solution. Let us denote by S'_{alg} one such valid solution. Then by step 9, it follows that $S'_{oracle} = ReverseSolutionMap(S'_{alg})$ is a valid solution for $I_{oracle} = InstanceMap(I_{alg})$. This witnesses the fact that I_{oracle} has a valid solution and hence I_{oracle} is an instance for which the answer is *Yes*. If Alg_{oracle} works correctly as supposed, then it returns *Yes* and a valid solution S_{oracle}. Our code for Alg_{alg} will then return the correct answer *Yes* and $S_{alg} = SolutionMap(S_{oracle})$, which by step 7 is a valid solution for I_{alg}.

No to No: We must now prove the reverse, that if the instance I_{alg} given to Alg_{alg} is a *No* instance, then Alg_{alg} answers *No*. The problem with *No* instances is that they have no witness to prove that they are *No* instances. Luckily, to prove something, it is sufficient to prove the contrapositive. Instead of proving $A \Rightarrow B$, where $A =$ "I_{alg} is a *No* instance" and $B =$ "Alg_{alg} answers *No*", we will prove that $\neg B \Rightarrow \neg A$, where $\neg B =$ "Alg_{alg} answers *Yes*" and $\neg A =$ "I_{alg} is a *Yes* instance." Convince yourself that this is equivalent.

If Alg_{alg} is given the instance I_{alg} and answers *Yes*, our code is such that Alg_{oracle} must have returned *Yes*. If Alg_{oracle} works correctly as supposed, the instance $I_{oracle} = InstanceMap(I_{alg})$ that it was given must be a *Yes* instance. Hence, I_{oracle} must have a valid solution. Let us denote by S_{oracle} one such valid solution. Then by step 7, $S_{alg} = SolutionMap(S_{oracle})$ is a valid solution for I_{alg}, witnessing I_{alg} being a *Yes* instance. This is the required conclusion $\neg A$.

This completes the proof that if the supposed algorithm Alg_{oracle} correctly solves P_{oracle}, then our algorithm Alg_{alg} correctly solves P_{alg}.

11) Running Time: The remaining step is to prove that the constructed algorithm Alg_{alg} runs in polynomial ($|I_{alg}|^{\Theta(1)}$) time. Steps 5 and 6 require that both $InstanceMap(I_{alg})$ and $SolutionMap(S_{oracle})$ work in polynomial time. Hence, if P_{oracle} can be solved "quickly", then Alg_{alg} runs in polynomial time. Typically for reductions people assume that Alg_{oracle} is an *oracle* meaning that it solves its problem in one time step. Exercise 0.2.5 explores the issue of running time further.

This concludes the proof that $P_{oracle} =$ Course Scheduling is NP-complete, (assuming of course that that $P_{alg} =$ 3-COL has already been proven to be NP-complete.)

Exercise 0.2.1 *We began this section by proving (Any Optimization Problem)* $\leq_{poly}$ *CIR-SAT. To make this proof more concrete redo it completing each of the above steps specifically for 3-COL* $\leq_{poly}$ *CIR-SAT. Hint: The circuit* $I_{oracle} = InstanceMap(I_{alg})$ *should have a variable* $x_{\langle u,c \rangle}$ *for each pair* $\langle u, c \rangle$.

Exercise 0.2.2 *3-SAT is a subset of the CIR-SAT problem in which the input circuit must be a big* AND *of clauses, each clause must be the* OR *of at most three literals, and each literal is either a variable or its negation. Prove that 3-SAT is NP-compete by proving that 3-COL* $\leq_{poly}$ *3-SAT. Hint: The answer is almost identical to that for Exercise 0.2.1.*

Exercise 0.2.3 *Let* $\overline{\text{CIR-SAT}}$ *be the complement of the CIR-SAT problem, namely the answer is* Yes *if and only if the input circuit is* not *satisfiable. Can you prove CIR-SAT* $\leq_{poly}$ $\overline{\text{CIR-SAT}}$ *using Cook reductions. Can you prove it using Karp reductions?*

Exercise 0.2.4 *Suppose problem* P_1 *is a restricted version of* P_2*, in that they are the same except* P_1 *is define on a subset* $\mathcal{I}_1 \subseteq \mathcal{I}_2$ *of the instances that* P_2 *is defined on. For example, 3-SAT is a restricted version of CIR-SAT because both determine whether a given circuit has a satisfying assignment, however, 3-SAT only considers special types of circuits with clauses of three literals. How hard is it to prove* $P_1 \leq_{poly} P_2$*? How hard is it to prove* $P_2 \leq_{poly} P_1$*?*

Exercise 0.2.5 *Suppose that when proving* $P_{alg} \leq_{poly} P_{oracle}$*, the routines* $InstanceMap(I_{alg})$ *and* $SolutionMap(S_{oracle})$ *each run in* $\mathcal{O}(|I_{alg}|^3)$ *time and that the mapping* $InstanceMap(I_{alg})$ *constructs from the instance* I_{alg} *an instance* I_{oracle} *that is much bigger, namely,* $|I_{oracle}| = |I_{alg}|^2$*. Given the following two running times of the algorithm* Alg_{oracle}*, determine the running time of the algorithm* Alg_{alg}*. (Careful.)*

1. $Time(Alg_{oracle}) = \Theta(2^{n^{\frac{1}{3}}})$
2. $Time(Alg_{oracle}) = \Theta(n^c)$ *for some constant c.*

0.3 Example: 3-Colouring is NP-Complete

We will now use the steps again to prove that 3-Colouring is NP-complete.

0) In NP: The problem 3-COL is in NP because given an instance graph I_{graph} and a solution colouring $S_{colouring}$, it is easy to have an algorithm $Valid(I_{graph}, S_{colouring})$ check that each node is coloured with one of three colours and that the nodes of each edge have different colours.

1) What To Reduce To It: We will reduce CIR-SAT to 3-COL by proving CIR-SAT $\leq_{poly}$ 3-COL. In Section 0.1 we proved (Any Optimization Problem) $\leq_{poly}$ CIR-SAT and that 3-COL $\leq_{poly}$ Course Scheduling. By transitivity, this gives us that CIR-SAT, 3-COL, and Course Scheduling are each NP-Complete problems.

2) What is What:

$\boldsymbol{P_{alg}}$ is the Circuit Satisfiability problem (CIR-SAT).

$\boldsymbol{I_{circuit}}$, an instance to it, is a circuit.

$\boldsymbol{S_{assignment}}$, a potential solution, is an assignment to the circuit variables $x_1, x_2, \ldots, x_n$.

$\boldsymbol{P_{oracle}}$ is the Graph 3-Colouring problem (3-COL).

$\boldsymbol{I_{graph}}$, an instance to it, is a graph.

$\boldsymbol{S_{colouring}}$, a potential solution, is an colouring of the nodes of the graph with 3 colours.

3) Direction of Reduction and Code: To prove 3-COL is "at least as hard", we must prove that CIR-SAT is "at least as easy", namely CIR-SAT $\leq_{poly}$ 3-COL. To do this, we must design an algorithm for CIR-SAT given an algorithm for 3-COL. The code will be identical to that in Section 0.2.

4) Look For Similarities: An assignment allocates *True/False* values to each variable, which in turn induces *True/False* values to the output of each gate. A colouring allocates one of three colours to each node. This similarity hints at mapping the variables and outputs of each gate to nodes in the graph and mapping *True* to one colour and *False* to another. With these ideas in mind, Steven Rudich had a computer search for the smallest graph that behaves like an *or* gate when coloured with three colours. The graph found is shown in Figure 1. He calls it an *OR* Gadget.

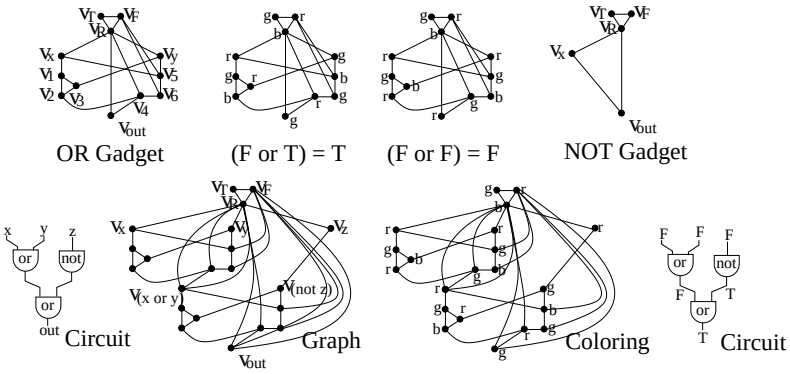


Figure 1: On the top, the first figure is the *OR* gadget. The next two are colourings of this gadget demonstrating $(False\ or\ True) = True$ and $(False\ or\ False) = False$. The top right figure is the *NOT* gadget. On the bottom, the first figure is the circuit given as an instance to SAT. The next is the graph that it is translated into. The next is a 3-colouring of this graph. The last is the assignment for the circuit obtained from the colouring.

Translating between Colours and True/False: The three nodes v_T, v_F, and v_R in the *OR* gadget are referred to as the *pallet*. Because of the edges between them, when the gadget is properly coloured, these nodes need to be assigned different colours. Whatever colour is assigned to the node v_T, we will say that it is the colour indicating *true*, that colouring v_F the colour indicating *false*, and that colouring v_R the *remaining* colour. For example, in all the colourings in Figure 1, green indicates *true*, red indicates *false*, and blue is the remaining colour.

Input and Output Values: Nodes v_x and v_y in the *OR* gadget act as the gadget's inputs and the node v_{out} as its output. Because each of these nodes has an edge to node v_R, they cannot be coloured with the *remaining* colour. The node will be said to "have the value *true*", if it is assigned the same colour as v_T and *false* if the same as v_F. The colouring in the second figure in Figure 1 sets $x = false$, $y = true$, and the output $= true$. The colouring in the third figure sets $x = false$, $y = false$, and the output $= false$.

Theorem 19.3.1: Rudish's *OR* gadget acts like an *or* gate, in that it always can be and always must be coloured so that the value of its output node v_{out} is the *or* of the values of its two input nodes v_x and v_y. Similarly for the *NOT* gate.

Proof: There are four input instances to the gate to consider.

(*False or True*) = *True*: If node v_x is coloured false and v_y is coloured true, then because v_5 has an edge to each, it must be coloured the remaining colour. v_6, with edges to v_F and v_5 must be coloured true. v_4, with edges to v_R and v_6 must be coloured false. v_{out}, with edges to v_R and v_4 must be coloured true. The colouring in the second figure in Figure 1 proves that such a colouring is possible.

(*False or False*) = *False*: If node v_x and v_y are both coloured false, then neither nodes v_1 nor v_3 can be coloured false. Because of the edge between them, one of them must be true and the other the remaining colour. Because v_2 has an edge to each of them, it must be coloured false. v_4, with edges to v_R and v_2 must be coloured true. v_{out}, with edges to v_R and v_4 must be coloured false. The colouring in the third figure in Figure 1 proves that such a colouring is possible.

(*True or True*) = *True* and (*True or False*) = *True*: See Exercise 0.3.1 for these cases and for the *NOT* gate.

5) InstanceMap, Translating the Circuit into a Graph: Our algorithm for the CIR-SAT takes as input a circuit $I_{circuit}$ to be satisfied and in order to receive help from the 3-COL algorithm constructs from it a graph $I_{graph} = InstanceMap(I_{circuit})$ to be coloured. See the first two figure on the bottom of Figure 1. The graph will have one pallet of nodes v_T, v_F, and v_R with which to define the true and the false colour. For each variable x_i of the circuit, it will have one node labeled x_i. It will also have one node labeled x_{out}. For each *or* gate and *not* gate in the circuit, the graph will have one copy of the *OR gadget* or the *NOT gadget*. The *and* gates could be translated into a similar *AND* gadget or translated to $[x \text{ and } y] = [not(not(x) \text{ or } not(y))]$. All of these gadgets share the same three pallet nodes. If in the circuit the output of one gate is the input of another then the corresponding nodes in the graph are the same. Finally, one extra edge is added to the graph from the v_F node to the v_{out} node.

6) SolutionMap, Translating a Colouring into an Assignment: When the supposed algorithm finds a colouring $S_{colouring}$ for the graph $I_{graph} = InstanceMap(I_{circuit})$, our algorithm must translate this colouring into an assignment $S_{assignment} = SolutionMap(S_{colouring})$ of the variables $x_1, x_2, \ldots, x_n$ for circuit. See the last two figures on the bottom of Figure 1. The translation is accomplished by setting x_i to true if node v_{x_i} is coloured the same colour as node v_T and false if the same as v_F. If node v_{x_i} has the same colour node v_R, then this is not a valid colouring because there is an edge in the graph from node v_{x_i} to node v_R and hence need not be considered.

Warning: Suppose that the graph constructed had a separate node for each time that the circuit used the variable x_i. The statement "set x_i to true when the node v_{x_i} has some colour" would then be ambiguous, because the different nodes representing x_i may be given different colours.

7) Valid to Valid: Here we must prove that if the supposed algorithm gives us a valid colouring $S_{colouring}$ for the graph $I_{graph} = InstanceMap(I_{circuit})$, then $S_{assignment} = SolutionMap(S_{colouring})$ is an assignment that satisfies the circuit. By the gadget theorem, each gadget in the graph must be coloured in a way that acts like the corresponding gate. Hence, when we apply the assignment to the circuit, the output of each gate will have the

value corresponding to the colour of corresponding node. It is as if the colouring of the graph is performing the computation of the circuit. It follows that the output of the circuit will have the value corresponding to the colour of node v_{out}. Because node v_{out} has an edge to v_R and an extra edge to v_F, v_{out} must be coloured true. Hence, the assignment is one for which the output of the circuit is true.

8) ReverseSolutionMap: For the proof we must also define the reverse mapping from each assignment $S_{assignment}$ to a colouring $S_{colouring} = ReverseSolutionMap(S_{assignment})$. Start by colouring the pallet nodes true, false, and the remaining colour. Colour each node v_{x_i} true or false according to the assignment. Then Theorem 19.3.1 states that no matter how the inputs nodes to a gadget is coloured, the entire gadget can be coloured with the output node having the colour as indicated by the output of the corresponding gate.

9) Reverse Valid to Valid: Now we prove that if the assignment $S_{assignment}$ satisfies the circuit, then the colouring $S_{colouring} = ReverseSolutionMap(S_{assignment})$ is valid. Theorem 19.3.1 ensured that each edge in each gadget has two different colours. The only edge remaining to consider is the extra edge. As the colours percolate down the graph node v_{out} must have colour corresponding to the output of the circuit, which must be the true colour because the assignment is satisfies the circuit. This ensures that even the extra edge from node v_F to v_{out} is coloured with two different colours.

10 & 11: These steps are always the same. $InstanceMap(I_{circuit})$ maps *Yes* circuit instances to *Yes* 3-COL instances and *No* to *No*. Hence, if the supposed algorithm 3-COL works correctly in polynomial time, then our designed algorithm correctly solves CIR-SAT in polynomial time. It follows that CIR-SAT $\leq_{poly}$ 3-COL. In conclusion, 3-Colouring is NP-complete.

Exercise 0.3.1 *(a) Complete the proof of Theorem 19.3.1 by proving the cases* $(True \text{ or } True) = True$ *and* $(True \text{ or } False) = True$. *(b) Prove a similar theorem for the* NOT *gadget. See the top right figure in Figure 1.*

Exercise 0.3.2 *Verify that each edge in the graph* $I_{graph} = InstanceMap(I_{circuit})$ *is needed by showing that if it was not there then it would be possible for the graph to have a valid colouring even when the circuit is not satisfied.*

Exercise 0.3.3 *(See solution in Section* **??***) Prove that Independent Set is NP-compete by proving that* $Time(3\text{-}COL) \leq Time(\text{Independent Set}) + n^{\Theta(1)}$. *Hint: A 3-coloring for the graph* G_{COL} *can be thought of as a subset of the pairs* $\langle u, c\rangle$ *where* u *is a node of* G_{COL} *and* c *is a color. An independent set of the graph* G_{Ind} *selects a subset of its nodes. Hence, a way to construct the graph* G_{ind} *in the instance* $\langle G_{Ind}, N_{Ind}\rangle = InstanceMap(G_{COL})$ *would be to having a node for each pair* $\langle u, c\rangle$. *The input to the i Be careful when defining the edges for the graph* $G_{Ind} = InstanceMap(G_{COL})$ *so that each valid independent set of size n in the constructed graph corresponds to a valid three coloring of the original graph. If the constructed graph has unexpected independent sets, you may need to add more edges to the graph.*

0.4 An Algorithm for Bipartite Matching using the Network Flow Algorithm

Up to now we have been justifying our belief that certain computational problems are difficult by reducing them to other problems believed to be difficult. Here, we will give an example of the

reverse, by proving that the problem *Bipartite Matching* can be solved easily by reducing it to the Network Flows problem, which we already know is easy because we gave an polynomial time algorithm for it in Section **??**.

Bipartite Matching: Bipartite matching is a classic optimization problem. As always, we define the problem by given a set of instances, a set of solutions for each instance, and a cost for each solution.

Instances: An input instance to the problem is a bipartite graph. A bipartite graph is a graph whose nodes are partitioned into two sets U and V and all edges in the graph go between U and V. See the first figure in Figure 2.

Solutions for Instance: Given an instance, a solution is a matching. A matching is a subset M of the edges so that no node appears more than once in M. See the last figure in Figure 2.

Cost of a Solution: The cost (or success) of a matching is the number of pairs matched. It is said to be a perfect matching if every node is matched.

Goal: Given a bipartite graph, the goal of the problem is to find a matching that matches as many pairs as possible.

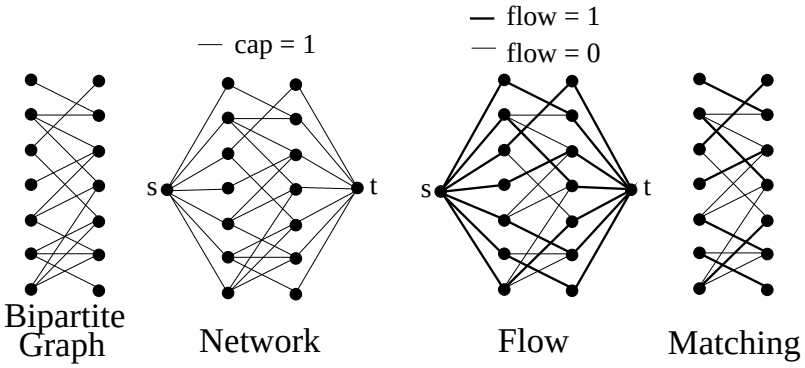


Figure 2: The first figure is the bipartite graph given as an instance to Bipartite matching. The next is the network that it is translated into. The next is a flow through this network. The last is the matching obtained from the flow.

Network Flows: Network Flow is another example of an optimization problem that involves searching for a best solution from some large set of solutions.

Instances: An instance $\langle G, s, t\rangle$ consists of a directed graph G and specific nodes s and t. Each edge $\langle u, v\rangle$ is associated with a positive capacity $c_{\langle u,v\rangle}$.

Solutions for Instance: A solution for the instance is a *flow* F which specifies a flow $F_{\langle u,v\rangle} \leq c_{\langle u,v\rangle}$ through each edges of the network with no leaking or additional flow at any node.

Measure Of Success: The cost (or success) of a flow is the the amount of flow out of node s.

Goal: Given an instance $\langle G, s, t\rangle$, the goal is to find an optimal solution, that is, a maximum flow.

Bipartite Matching $\leq_{poly}$ Network Flows: We go through the same steps as before.

3) Direction of Reduction and Code: We will now design an algorithm for Bipartite Matching given an algorithm for Network Flows.

4) Look For Similarities: A matching decides which edges to keep and a flow decides which edges to put flow though. This similarity suggests keeping the edges that have flow through them.

5) InstanceMap, Translating the Bipartite Graphs into a Network: Our algorithm for Bipartite Matching takes as input a bipartite graph $G_{bipartite}$. The first step is to translate this into a network $G_{network} = InstanceMap(G_{bipartite})$. See the first two figures in Figure 2. The network will have the nodes U and V from the bipartite graph and for each edge $\langle u, v\rangle$ in the bipartite graph, the network has a directed edge $\langle u, v\rangle$. In addition, the network will have a source node s with a directed edge from s to each node $u \in U$. It will also have a sink node t with a directed edge from each node $v \in V$ to t. Every edge out of s and every into t will have capacity one. The edges $\langle u, v\rangle$ across the bipartite graph could be given capacity one as well, but they could just as well be given capacity ∞.

6) SolutionMap, Translating a Flow into an Matching: When the Network Flows algorithm finds a flow S_{flow} through the network, our algorithm must translate this flow into a matching $S_{matching} = SolutionMap(S_{flow})$. See the last two figures in Figure 2.

SolutionMap: The translation puts the edge $\langle u, v\rangle$ in the matching if there is a flow of one through the corresponding edge in the network and not if there is no flow in the edge.

Warning: Be careful to map *every* possible flow to a matching. The above mapping is ill defined when there is a flow of $\frac{1}{2}$ through an edge. This needs to be fixed and could be quite problematic.

Integer Flow: Luckily, Exercise **??** proves that if all the capacities in the given network are integers, then the algorithm always returns a solution in which the flow through each edge is an integer. Given that our capacities are all one, each edge will either have a flow of zero or of one. Hence, in our translation, it is well-defined whether to include the edge $\langle u, v\rangle$ in the matching or not.

7) Valid to Valid: Here we must prove that if the flow S_{flow} is valid than the matching $S_{matching}$ is also valid.

Each u Matched At Most Once: Consider a node $u \in U$. The flow into u can be as most one because there is only one edge into it and it has capacity one. For the flow to be valid, the flow out of this node must equal that in. Hence, it too can be at most one. Because each edge out of u either has flow zero or one, if follows that at most one edge out of u has flow. We can conclude that u is matched to at most one node $v \in V$.

Each v Matched At Most Once: See Exercise 0.4.1

Cost to Cost: To be sure that the matching we obtain contains the maximum number of edges, it is important that the cost of the matching $S_{matching} = SolutionMap(S_{flow})$ equals the cost of the flow. The cost of the flow is the amount of flow out of node s, which equals the flow across the cut $\langle U, V\rangle$, which equals the number of edges $\langle u, v\rangle$ with flow of one, which equals the number of edges in the matching, which equals the cost of the matching.

8) ReverseSolutionMap: The reverse mapping from each matching $S_{matching}$ to a valid flow $S_{flow} = ReverseSolutionMap(S_{matching})$ is straight forward. If edge $\langle u, v\rangle$ is in the matching, then put a flow of one from the source s, along the edge $\langle s, u\rangle$ to node u, across the corresponding edge $\langle u, v\rangle$, and then on through the edge $\langle v, t\rangle$ to t.

9) Reverse Valid to Valid: We must also prove that if the matching $S_{matching}$ is valid then the flow $S_{flow} = ReverseSolutionMap(S_{matching})$ is also valid.

Flow in Equals Flow Out: Because the flow is the sum of paths, we can be assured that the flow in equals the flow out of every node except for the source and the sink. Because the matching is valid, each u and each v is matched either zero or once. Hence the flows through the edges $\langle s, u\rangle$, $\langle u, v\rangle$, and $\langle v, t\rangle$ will be at most their capacity one.

Cost to Cost: Again, we need to prove that the cost of the flow $S_{flow} = ReverseSolutionMap(S_{matching})$ is the same as the cost of the matching. See Exercise 0.4.2.

10 & 11: These steps are always the same. $InstanceMap(G_{bipartite})$ maps bipartite graph instances to network flow instances G_{flow} with the same cost. Hence, because algorithm Alg_{flow} correctly solves network flows quickly, our designed algorithm correctly solves bipartite matching quickly.

In conclusion, bipartite matching can be solved in the same time that network flows is solved.

Exercise 0.4.1 *Give a proof for the case where each v is matched at most once.*

Exercise 0.4.2 *Give a proof that the cost of the flow $S_{flow} = ReverseSolutionMap(S_{matching})$ is the same as the cost of the matching*

Exercise 0.4.3 *Section* **??** *constructs three dynamic programming algorithms using reductions. For each of these, carry out the formal steps required for a reduction.*

Exercise 0.4.4 *There is a collection of software packages $S_1, \ldots, S_n$ which you are considering acquiring. For each i, you will gain an overall benefit of b_i if you acquire package S_i. Possibly b_i is negative, for example, if the cost of S_i is greater than the money that will be saved by having it. Some of these packages rely on each other; if S_i relies on S_j, then you will incur an additional cost of $C_{i,j} \geq 0$ if you acquire S_i but not S_j. Unfortunately, S_1 is not available. Provide a polytime algorithm to decide which of $S_2, \ldots, S_n$ you should acquire. Hint: Use max flow / min cut.*