CS Research Methodology 2011
National University of Singapore CS2309
A/P Lee Wee Sun

Proofs

1 Proof Techniques

1.1 Terminology

We begin with commonly used terminology.

- **Statement**: A statement is a sentence that is either true or false. For example, "Two plus two equals four" or "If a and b are both even, then $a + b$ is even.
- **Hypothesis**: A hypothesis is a statement that is assumed to be true. For example, in "If a and b are both even, then $a + b$ is even", the part "a and b are both even" is the hypothesis.
- **Conclusion**: A conclusion is a statement that follows from the hypotheses. For example, in "If a and b are both even, then $a + b$ is even", the part "$a + b$" is the conclusion.
- **Definition**: The precise meaning of a word, phrase, mathematical symbol or concept that ends all confusion.
- **Proof**: A logical argument that establish truth of a statement beyond doubt. A proof is a chain of steps, each a logical consequence of the previous one.
- **Theorem**: A mathematical statement whose truth can be established by the assumptions given (or implied) in the statement.
- **Lemma**: An auxiliary theorem proven so that it can be used to prove another theorem.
- **Corollary**: A result that follows easily from a theorem already proved.
- **Validity**: An argument is valid if the hypotheses supplies a sufficient basis for the conclusion to be reached.
  - An argument can be valid but reach a false conclusion if one of the hypotheses is false e.g. *Penguins are birds. All birds are able to fly. Therefore penguins are able to fly.*
  - An argument can be invalid but reach a true conclusion e.g. *Giraffes have four legs. Cows have four legs. Therefore giraffes are taller than cows.*
- A **sound** proof uses valid arguments.

We will look mainly at proving statements of the type: “If A then B” e.g. ”If an animal is a cow, then it has four legs”. For this type of statement to be true we need to know that whenever A is true B has to be true as well. If A is false, it does not matter whether B is true or false. This is often reworded as “A is a **sufficient condition** for B”, or as “B is a **necessary condition** for A”.

These types of statements can be used as a type of valid argument known as **modus ponens**, i.e. given that the statement is true, if we know A is true, we can conclude B is true.

Often, we hear about converse and contrapositive of a statement. The **converse** of the statement “If A then B” is “If B then A”. The **contrapositive** of the statement “If A then B” is “If not B then not A”

The contrapositive is particularly useful as it is equivalent to the original statements. We can check this by using the truth table.

A	B	$\neg B$	$\neg A$	$A \to B$	$\neg B \to \neg A$
F	F	T	T	T	T
F	T	F	T	T	T
T	F	T	F	F	F
T	T	F	F	T	T

1.2 Direct Proof

Usually, we give a direct proof of a statement. In a direct proof, we use the information in the hypothesis to construct a series of logical steps that leads to the conclusion. For example:

Theorem 1 *The sum of the first n natural numbers is equal to $n(n+1)/2$.*

Let us first rewrite the statement as *if A then B*:

- A: the first n natural numbers are summed together
- B: the sum is $n(n+1)/2$.

A useful thing to do first is to try a few small instances. For example $1 + 2 + 3 = 6$ agrees with the formula.

Proof. We will directly construct the formula. Let

$$S_n = 1 + 2 + 3 + \cdots + n.$$

This can also be written as

$$S_n = n + (n-1) + \cdots + 2 + 1$$

We have exploited the fact that summation is *invariant* to permutation of the elements. Summing up the two equations, we get

$$2S_n = (1+n) + [2 + (n-1)] + \cdots + [(n-1) + 2] + (n+1)$$

or $2S_n = n(n+1)$, giving our desired outcome of $S_n = n(n+1)/2$. □

1.3 Proof by Contradiction

This is one of the most common proof technique, also known as proving the contrapositive or using an indirect proof. In this technique, we want to prove "If P then Q". Instead of doing a direct proof, we assume the negation, $\neg Q$, then show $\neg P$. In other words, we assume that the conclusion is false and show that a contradiction with the hypothesis is reached. Note that this is the same as proving the contrapositive "If $\neg Q$ then $\neg P$". We give a simple example below.

Theorem 2 *There is an infinite number of primes.*

Proof. Assume that there is a finite number of primes. Then there must be a last prime P (look at an *extreme* point). We now construct a number Q that is larger than P

$$Q = (2 \times 3 \times 5 \times 7 \times 11 \times \ldots P) + 1.$$

The number Q is greater than P, so cannot be prime (by assumption that P is largest prime). Consequently, Q must be divisible by a prime. Let us check by dividing by all primes. In all cases, the remainder is 1, showing that Q must be prime. This contradicts our assumption that P is the largest prime. Hence, our assumption that there is a finite number of prime must be wrong. $\square$

Note that we in the theorem above we are proving "If P then Q"where Q is the statement "there is an infinite number of primes" but P is not stated. In such cases, P is taken as the usual laws of mathematics. In the proof above, we assumed $\neg Q$ and obtained a contradiction in the law of mathematics.

We give another more complex example here.

Halting Problem: You are an employee of the software company that is writing a new operating system. Your boss wants you to write a program that will take in a user's program and inputs and decide whether

- it will eventually stop, or
- it will run infinitely in some infinite loop.

If the program will run infinitely, your program will disallow the program to run and send the user a rude message.

Theorem 3 *It is not possible to write a program that*

- *takes in a program P and an input I*
- *runs for a finite amount of time and correctly answers whether the program P will halt on input I.*

We give a proof sketch here, assuming your usual understanding of what a program can and cannot do.

Proof Sketch. Assume that it is possible to write a program to solve the Halting Problem. Denote this program by HALTANSWERER(PROG, INPUTS). The program HALTANSWERER(PROG, INPUTS) will return

- *yes* if PROG will halt on INPUTS and
- *no* otherwise.

Note that a program is just a string of characters e.g. your Java program is just a long string of characters. An input can also be considered as just a string of characters. So HALTANSWERER is just a computer program that takes two strings as inputs.

We can now write another program NASTY(PROG) that uses HALTANSWERER as a subroutine. The program NASTY(PROG) does the following:

1. If HALTANSWERER(PROG, PROG) returns *yes*, NASTY will go into an infinite loop.
2. If HALTANSWERER(PROG, PROG) returns *no*, NASTY will halt.

Consider what happens when we run NASTY(NASTY).

- If NASTY loops infinitely, HALTANSWERER(NASTY, NASTY) return *no* which by (2) above means NASTY will halt.
- If NASTY halts, HALTANSWERER(NASTY, NASTY) will return *yes* which by (1) above means NASTY will loop infinitely.

Hence, our assumption that it is possible to write a program to solve the Halting problem has resulted in a contradiction showing that such a program is not possible. □

In the theorem, we have contradicted the fact that a program cannot both halt and loop infinitely.

1.4 Principle of Induction

Use of induction, invariance and reduction are probably the most common proof techniques in computer science.

Mathematical induction is very powerful method for proving assertions that are indexed by integers, for example: $n! > 2^n$ for all positive integers $n \geq 4$.

The assertions in this example be put in the form: $P(n)$ is true for all integers $n \geq n_0$. Mathematical induction works on such assertions. To do a proof by induction, you need to:

- Establish the truth of $P(n_0)$. This is called the base case, and is usually easy to prove.
- Assume that $P(n)$ is true for some arbitrary integer $n \geq n_0$, or for all integers m with $n_0 \leq m \leq n$. This is called the **inductive hypothesis**.
- Then show that the inductive hypothesis implies that $P(n+1)$ is also true.

Example: Program correctness by induction. Algorithm 1 is a binary search program that returns whether x is between elements indexed by a and b in array A. We would like to use mathematical induction to prove the correctness of the program.

Algorithm 1 BINARYSEARCH(A, a, b, x)

```
if $a > b$ then
  return no
else
  $\text{MID} = \lfloor (a + b)/2 \rfloor$
end if
if $x = A[\text{MID}]$ then
  return yes
else if $x < A[\text{MID}]$ then
  BINARYSEARCH$(A, a, \text{MID} - 1, x)$
else
  BINARYSEARCH$(A, \text{MID} + 1, b, x)$
end if
```

To prove correctness, we specify **precondition** and **postcondition**.

- The precondition states what may be assumed to be true initially:
  Pre: $a \leq b + 1$ and $A[a..b]$ is a sorted array
- The postcondition states what is to be true about the result
  found = BINARYSEARCH(A, a, b, x);
  Post: $found = x \in A[a..b]$ and A is unchanged

The proof takes us from the precondition to the postcondition. In the base case, we have array size $n = b - a + 1 = 0$. To see that the program works correctly for $n = 0$, observe that

- The array is empty, so $a = b + 1$.
- This allow the test $a > b$ to succeed and the algorithm correctly returns false.

Now consider the inductive step: $n = b - a + 1 > 0$. For the inductive hypothesis, assume BINARYSEARCH(A, a, b, x) returns the correct value for all j such that $0 \leq j \leq n - 1$ where $j = b - a + 1$.

- The algorithm first calculates $\text{MID} = \lfloor (a + b)/2 \rfloor$, thus $a \leq \text{MID} \leq b$.
- If $x = A[\text{MID}]$, clearly $x \in A[a..b]$ and the algorithm correctly returns true.
- If $x < A[\text{MID}]$, since A is sorted (by the precondition), x is in $A[a..b]$ if and only if it is in $A[a..\text{MID} - 1]$.

- By the inductive hypothesis, $BinarySearch(A, a, \text{MID} - 1, x)$ will return the correct value since $0 \leq (mid - 1) - a + 1 \leq n - 1$.

- The case $x > A[\text{MID}]$ is similar.

We have shown that the postcondition holds if the precondition holds and BinarySearch is called. Hence the program works correctly.

The following example illustrates an interesting phenomenon with proofs by induction: sometimes it is easier to prove a stronger statement.

Example: Divide and Conquer Recurrence. Assume that you have a divide and conquer algorithm that divides a problem of size n into four parts of size $n/2$, solves the four subproblems and then combine them. Assume that the total time for doing the divide and combine steps is $O(n)$. Compute the asymptotic running time.

We can write down the *recurrence* for the running time for a problem of size n:

$$T(n) \leq 4T(n/2) + Cn.$$

We first guess that the running time is $T(n) \leq C_1 n^2$ and try to do a proof by induction. Using $T(k) \leq C_1 k^2$ for $k < n$ as the induction hypothesis, we have:

$$\begin{aligned} T(n) &\leq 4T(n/2) + Cn \\ &\leq 4C_1(n/2)^2 + Cn \\ &= C_1 n^2 + Cn. \end{aligned}$$

Unfortunately, we cannot continue as there is no positive constant C_1 for which $C_1 n^2 + Cn \leq C_1 n^2$. The trick is to try to prove something stronger i.e. $T(n) \leq C_1 n^2 - C_2 n$. The inductive hypothesis is now $T(k) \leq C_1 k^2 - C_2 k$ for $k < n$, giving:

$$\begin{aligned} T(n) &\leq 4T(n/2) + Cn \\ &\leq 4C_1(n/2)^2 - 4C_2 n/2 + Cn \\ &= C_1 n^2 - (2C_2 n - Cn) \\ &\leq C_1 n^2 - C_2 n \end{aligned}$$

when $C_2 \geq C_1$. To complete the proof, we need to find some n_0 where $T(n_0) \leq C_1 n_0^2 - C_2 n_0$. This can be done by selecting C_1 large enough.

Why is it easier to prove the stronger result here? In a proof by induction, proving the stronger result also means that we can make a stronger induction hypothesis, which makes it easier to prove the inductive step. Of course, the result you are trying to prove must be true, e.g. you would not be able to prove that $T(n) \leq C_1 n$ for the recurrence above.

1.5 Using Invariants

Invariance, things that do not change, is a generally useful concept to keep in mind when solving problems. It is a particularly useful concept for analyzing iterative programs.

The key step in the proof is the invention of a condition called the **loop invariant**, which is supposed to be true at the beginning of an iteration and remains true at the beginning of the next iteration. The steps required to prove the correctness of an iterative algorithms is as follows:

1. Guess a condition I
2. Prove by induction that I is a loop invariant
3. Prove that $I \wedge \neg G \Rightarrow$ Postcondition
4. Prove that the loop is guaranteed to terminate

We give an example showing how it is used to prove correctness of a simple program.

Example: Summing an array. Given an array of numbers $A[a..b]$ of size $n = b - a + 1 \geq 0$, compute their sum.

```
Pre: $a \leq b + 1$
$i \leftarrow a$, $\textsc{sum} \leftarrow 0$
while $i \neq b + 1$ do {exit condition, called guard $G$}
  $\textsc{sum} \leftarrow \textsc{sum} + A[i]$
  $i \leftarrow i + 1$
end while
Post: $\textsc{sum} = \sum_{j=a}^{b} A[j]$
```

In the example, we know that when the algorithm terminates with $i = b + 1$, the following condition needs to hold: $\textsc{sum} = \sum_{j=a}^{i-1} A[j]$. Use that as the loop invariant.

We need to show that at the begining of the k-th loop, the loop invariant holds. We will do so by induction.

Base Case: $k = 1$. Initialized to $i = a$ and $\textsc{sum} = 0$. Therefore $\sum_{j=a}^{i-1} A[j] = 0$.

Inductive Hypothesis: Assume $\textsc{sum} = \sum_{j=a}^{i-1} A[j]$ at the start of the loop's k-th execution. Let $\textsc{sum}'$ and i' be the values of the variables $\textsc{sum}$ and i at the beginning of the $(k + 1)$st iteration. In the k-th iteration, the variables were changed as follows: $\textsc{sum}' = \textsc{sum} + A[i]$ and $i' = i + 1$. Using the inductive hypothesis we have

$$\textsc{sum}' = \textsc{sum} + A[i] = \sum_{j=a}^{i-1} A[j] + A[i] = \sum_{j=a}^{i} A[j] = \sum_{j=a}^{i'-1} A[j].$$

We have proven the loop invariant I. Now we show that $I \wedge \neg G \Rightarrow$ Postcondition. We have $\neg G \Rightarrow i = B + 1$. Substituting into the invariant:

$$\textsc{sum} = \sum_{j=a}^{b+1-1} A[j] = \sum_{j=a}^{b} A[j] \equiv \text{Postcondition}$$

It remains to be shown that G will eventually be false. Note that i is monotonically increasing since it is incremented inside the loop and not modified elsewhere. From the precondition, i is initialized to $a \leq b+1$, hence it will eventually be equal to $b+1$.

Here's another example.

Example: Coloring Planes. Consider a set of n (infinitely long) lines. These lines partition the plane into a finite number of regions. Show that the regions can be colored using two colors such that any two regions with a common boundary has a different color.

Algorithm 2 PLANECOLORING$(P, (l_1, \ldots, l_n))$

```
The entire plane $P$ is colored white
$i = 1$
while $i \neq n + 1$ do
  Cut the plane into half using line $l_i$
  On one half, keep the coloring the same
  On the other half, white is switched to black and black is switched to white.
  $i = i + 1$
end while
```

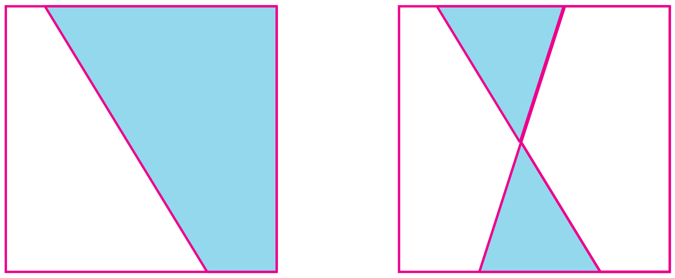

Figure 1: After the first and second lines are added.

We will prove that algorithm PLANECOLORING achieves the coloring. The precondition and postcondition are:

- Pre: The lines $l_1, \ldots, l_n$ specifies $n \geq 0$ infinitely long lines.
- Post: The plane P is colored using two colors such that any two regions that share a boundary have a different color.

The loop invariant is: The plane P partitioned with lines $l_1, \ldots, l_{i-1}$ is a proper coloring i.e. any two regions that share a boundary have a different color.

Before the loop is entered, the invariant is true as the plane is colored with one color and is not partitioned by any line. Hence the base case is true.

Assume that the invariant is true before the kth loop (inductive hypothesis). Within the loop, add the ith line and keep the colors on one side of the new line the same while flipping the colors on the other side of the plane. As a result, boundaries formed by the ith line have different colors on each side. By the inductive hypothesis, all boundaries formed by earlier lines have different colors on each side and this is not changed by flipping all the colors on one side of the new line. The variable i is incremented by 1 in the loop. Hence, all boundaries have different colors on each side after being partitioned by $l_1, \dots, l_i$ after the kth loop maintaining the loop invariant.

The loop terminates when $i = n + 1$, hence the postcondition is implied by the loop invariant and the guard condition. Finally, i is initialized to 1 and incremented by 1 in each iteration and so will eventually reach the value $n + 1$, ensuring termination.

1.6 Reduction

A reduction is a transformation from one problem to another problem. Reductions are often used to exploit a known method for solving a problem. For example, often we reduce a problem to a SAT problem because we have a good SAT solver. Reductions also have another use: showing that one problem is as difficult as another problem.

Example: Completely Automated Public Turing test to tell Computers and Humans Apart (Captcha). How do you ban robot web crawler from filling in forms and accessing websites that you would like to reserve for humans. Captcha relies on the inability of computers to solve some problems that humans are able to solve. Examples of these problems include recognizing ambiguously written characters (Figure 2). Effectively, it is reducing the difficult computational problem (e.g. the computer vision problem) to the problem of automatically filling form or accessing websites. So automatically filling in forms becomes as difficult as the difficult computational problem. As long as the computational problem has not been solve we would expect that robot crawlers are not able to automatically fill the forms in.

The example shows an engineering use of the reducing a difficult problem to the problem you are interested in. This idea is often used to show that a problem is formally difficult to solve, as in the theory of NP-completeness.

1.6.1 NP-Completeness

A decision problem is in the class NP (non-deterministic polynomial) if it is possible to **verify** the solution in polynomial time. More precisely, a problem is in NP if there exists an verifying algorithm V that takes in encoding of an instance I of the problem, and a proposed solution S such that

- A runs in time polynomial in $|I|$, and
- S is a solution for the instance I if and only if $V(I, S) =$ true.

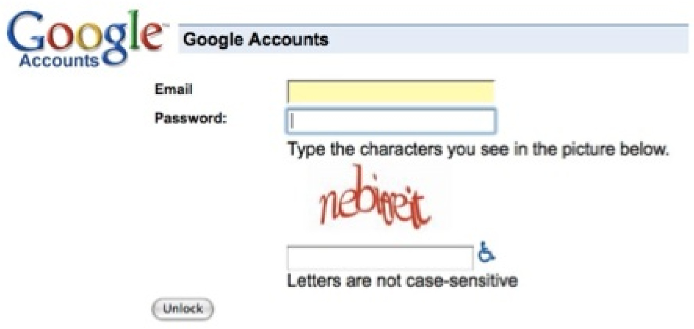
Google Accounts
Google Accounts
Email
Password:
Type the characters you see in the picture below.
nebiwrit
Letters are not case-sensitive
Unlock

Figure 2: Example of a captcha to ensure that user is human.

The class NP includes many search problems of practical interest including (decision versions of) all the graph problems we have discussed.

In contrast, the class of all problems that can be *solved* (instead of verified) in polynomial time is the class P. The class P is a subset of NP as the algorithm that solves the problem can also be used to verify the solution.

NP-complete problems are the *hardest* problems in NP in the sense that if we can solve an NP-complete problem in polynomial time, we can also solve *all* problems in NP in polynomial time. A problem A is NP-complete if

- It is in NP.
- Every problem in NP can be reduced to the problem in time polynomial in the size of the original problem. More precisely, given a problem B, there exists an algorithm R that transforms any instance I in B into an instance I' in A in time polynomial in $|I|$, such that the answer to I (as a problem in A) is *true* if and only if the answer to I' (as a problem in B) is *true*.

We have seen that SAT can be used to represent many problems. In fact, Cook's theorem says that every problem in NP can be represented as a polynomial sized SAT problem.

Theorem 4 (Cook's Theorem) *SAT is NP-complete.*

Proof Sketch. By definition, it takes a polynomial number of steps to verify the solution of a problem in NP.

A computer is essentially a combinational circuit (using AND, OR and NOT gates). Since it takes a polynomial number of steps to verify the solution, we can construct a circuit for verifying the solution for a problem in NP by constructing polynomial number of copies of the computer's circuit, one for each step, with the output of one time step used as input of the next time step.

The inputs to this circuit are the inputs to the verifying algorithm, which includes the bits to the solution S that is being verified. Knowing whether there exists a setting of the bits to the inputs corresponding to S would tell us the answer to the problem whose solution we constructed the circuit to verify. This show that the problem of whether there is an input that will make the output of a combinational circuit *true* is NP-complete. Finally, it is possible to reduce the description of a combinational circuit to a CNF of approximately the same size, showing that SAT is NP-complete. $\square$

To show that a target problem is NP-complete, we need to show a polynomial time transformation (reduction) from a known NP-complete problem into the target problem such that solving the target problem will also solve the NP-complete problem (NP-hardness). If the target problem is also in NP (solution can be verified quickly), then it is also NP-complete. SAT for CNF has been shown to be NP-complete (Cook) and can be used for reduction. It can be shown that 3-CNF-SAT (SAT for CNF formulae where each clause has exactly 3 literals such as $(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z)$) is also NP-complete by reduction from SAT.

CLIQUE A clique in a graph $G = (V, E)$ is a subset of vertices V such that for each pair u, v of vertices in V, (u, v) is an edge in E i.e. V is a complete subgraph in V. The CLIQUE problem asks whether there is a clique of size k in a graph.

Theorem 5 *CLIQUE is NP-complete.*

Proof. Given a set of k vertices, we can check if every pair of vertices is in the list of edges in polynomial time. Hence the problem is in NP. For each clause C_i, create three nodes corresponding to the literals in the clauses. For example, the clause $C_i = (x \vee \neg y \vee z)$ would have three nodes v^i_x, $v^i_{\neg y}$ and v^i_z corresponding to the literals x, $\neg y$ and z within it. Hence if we have m clauses, we will have $3m$ nodes. Two nodes are connected by an edge if they come from different clauses and their literals are not the negation of each other. We need to construct a total of $3m$ nodes and check all pairs of nodes to see if it should be connected by an edge. Time is hence polynomial in m.

We will now show that the question of whether a 3-CNF formula with m clauses has a satisfying assignment is equivalent to whether there is a clique of size m in the corresponding graph. We do that by showing that

- If the CNF formula has a satisfying assignment, then the graph has a clique of size m, and
- If the CNF formula does not have a satisfying assignment, then the graph does not have a clique of size m.

Assume that a 3-CNF formula with m clauses has a satisfying assignment.

- Each clause has at least one literal assigned 1.
- We pick one of these corresponding nodes from each clause.

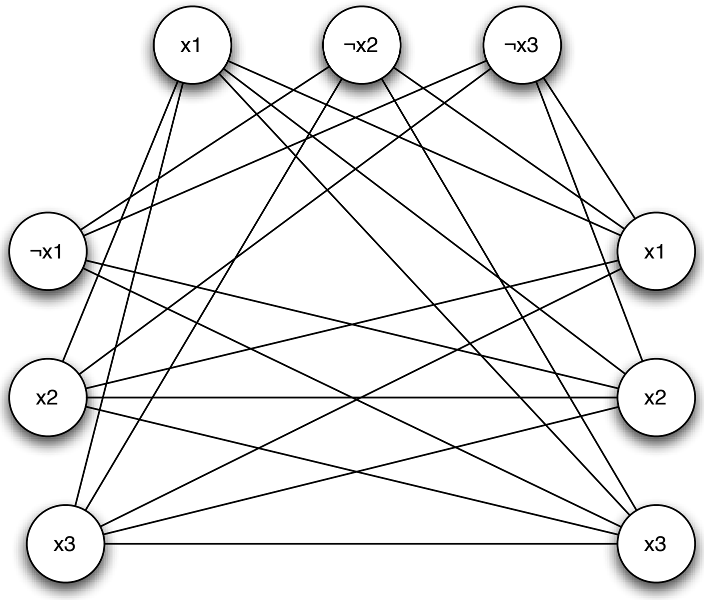

Figure 3: Graph derived from $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_{\vee} x_3) \wedge (x_1 \vee x_2 \vee x_3)$

By construction, each of these nodes is connected to every other selected node as the literals cannot be complements and they come from different clauses. Hence, there is a clique of size m in the corresponding graph.

We now argue that if the CNF formula does not have a satisfying assignment, then the graph does not have a clique of size m. We do this by showing the contrapositive of the previous statement (recall that a statement and its contrapositive are equivalent): if the corresponding graph G has a clique of size m, the formula has a satisfying assignment. No edges in the graph connect vertices corresponding to the same clause, hence the nodes in the clique must come from different clauses. The literals corresponding to these nodes can be assigned 1 as they are not complement of each other. Other variables not corresponding to vertices in the clique can be set arbitrarily to get an assignment that satisfies the formula. $\Box$

Theorem 6 *The problem of whether there is an independent set of size k in a a graph is NP-complete.*

Proof. We check whether there is an independent set of size k by reduction from CLIQUE. To check whether a set is an independent set, we only need to check that all vertices connected to each vertex in the set is not also in the set. This takes polynomial time, hence the problem is in NP. Given a CLIQUE problem for the graph $G = (V, E)$ form the complement graph $\bar{G} = (V, \bar{E})$ where $(u, v) \in \bar{E}$ if and only if $(u, v) \notin E$. Assume that there is a clique of size k in G. Then there is an independent set of size k in $\bar{G}$ as any two nodes in the clique in G is not connected with an edge in $\bar{G}$. Assume there is an independent set of size of size k in $\bar{G}$. Then there is a clique of size k in G as any two nodes in the independent set in $\bar{G}$ must be connected in G. $\Box$

2 Sanity Checks

Making mistakes in proofs is common. Here are some common ways to detect errors.

The statement that you are trying to prove may turn out to be false. Often, proving a statement may be difficult but it is easy to find a counterexample to show that the statement is false. For example, assume the claim

$$(a+b)^2 = a^2 + b^2.$$

We can easily show that the claim is false by checking values for $a = 1, b = 2$. You should at least try out a few small examples to make sure that counterexamples are not easy to find.

A counterexample need not apply to only the statement of a theorem, but may also apply to the proof method. If your proof method for showing that $P \neq NP$ also prove that 2-SAT is not in P, you should be very suspicious of your proof method, as 2-SAT is known to be in P.

Other than a counterexample to your theorem, it may turn out that your theorem solves known difficult problems. For example, if a proof that your algorithm runs in polynomial time also means that an NP-complete problem can be solved in polynomial time, you should be very suspicious of your proof.

Counterexamples often provide insights into how to construct a proof of a corrected statement. In fact Lakatos [3] argued that discoveries in mathematics are similar to discoveries in natural sciences - mathematicians make conjectures (or state erroneous theorems) which are refuted by counterexamples, which then led to more refined corrected theorems.

3 Exercise

1. The pigeonhole principle states that: If there are $m > n$ pigeons in n holes, there must be at least one hole with more than one pigeon. Give a proof to the pigeonhole principle using proof by contradiction.

2. Consider the recurrence $T(n) = 2T(n/4)+c$. Prove that $T(n) = O(\sqrt{n})$ using mathematical induction.

3. Consider the statement: All students have the same height. We will prove it by induction on the proposition: In any group of students of size n, all students have the same height.

    - **Base Case:** If there is one student in the group, obviously the student is of one height.
    - **Inductive step:** Assume that every groups of n students have the same height. Now consider a set of $n + 1$ student: $s_1, \ldots, s_n, s_{n+1}$. Students $s_1, \ldots, s_n$ have the same height as they belong to a group of n students. Similarly, students $s_2, \ldots, s_{n+1}$ have the same height as they belong to a group of n students. Therefore, any group of students of size $n + 1$ also have the same height.

    What is wrong with the proof?

4. Consider the bubblesort algorithm.

Algorithm 3 BUBBLESORT(A)

```
for $i = 1$ to LENGTH$[A]$ do
  for $j =$ LENGTH$[A]$ downto $i + 1$ do
    if $A[j] < A[j - 1]$ then
      swap $A[j]$ with $A[j - 1]$
    end if
  end for
end for
```

   Give a proof of correctness using a loop invariant.

5. Give a dynamic programming algorithm for finding the length of the longest monotonically increasing subsequence in a sequence of n numbers (from last set of exercises). For example, in the sequence S=(9,5,2,8,7,3,1,6,4), the longest increasing subsequence has length 3 and is either (2,3,4) or (2,3,6). Now, prove the correctness of the algorithm using a loop invariant.

6. The Goldbach Conjecture states that all even integers greater than 2 can be written as the sum of two primes. It is one of the oldest unsolved problem. It is easy to write a program that iterates through all integers, test whether it is the sum of two smaller primes and halts when it finds counterexample. If the program HALTANSWERER exists, we can submit this program to HALTANSWERER with no input to get the answer to the Goldbach Conjecture. What does this mean with regards to the relative difficulty of the Goldbach Conjecture and the Halting problem?

7. A Hamiltonian cycle of an undirected graph is a simple cycle (no repeated node) that contains each vertex in the graph. The problem of deciding whether a graph contains a Hamiltonian cycle is NP-complete. In the traveling salesman problem, you are given a complete weighted graph and a number k, and have to decide whether there exists a tour with a cost of at most k. Show that the traveling salesman problem is NP-complete by reduction from the Hamiltonian cycle problem.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms.

[2] A. Cupillari. *The nuts and bolts of proofs*. Academic Pr, 2001.

[3] I. Lakatos. Proofs and refutations: The logic of mathematical discovery (J. Worrall & E. Zahar, Eds.). *Cambridge, UK: Cambridge University*, 1976.