CS2309 CS Research Methodology

Proofs

Lee Wee Sun
School of Computing
National University of Singapore
leews@comp.nus.edu.sg

Semester 1, 2011/12

Outline

- Proof Techniques
    - Terminology
    - Direct Proof
    - Proof by Contradiction
    - Induction Principle
    - Invariant Principle
    - Reduction Principle

Terminology

- Statement: A statement is a sentence that is either true or false. 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. 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. 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 at the statement of the type “If A then B” e.g. ”If an animal is a cow, then it has four legs”.

- This type of statement is true whenever we 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”.
- Arguments of this form, called modus ponens are valid i.e. given that the statement is true, if we know A is true, we can conclude B is true.

We will look at a few ways to prove that this type of statement is true.

Related Statements

- 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 original statement is equivalent to the contrapositive.

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

Direct Proof

In a direct proof, we use the information in the hypothesis to construct a series of logical steps that leads to the conclusion.

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

Rewrite 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 $S_n = n(n+1)/2$.
This proof is known as Gauss' proof. □

Proof by Contradiction (Contrapositive, or Indirect Proof)

- Want to prove "If P then Q."
- Recall equivalence: If $\neg Q$, then $\neg P$.
- Assume conclusion false, show that a contradiction with the hypothesis is reached.

Infinitely Many Primes

Show that there is an infinite number of primes.

- Assume that there is a finite number of primes.
- Then there must be a last prime p (look at an extreme point)
- Aim: Construct a number larger than p that has remainder when divided by all smaller numbers $q = (2 \times 3 \times 5 \times 7 \times 11 \times \ldots P) + 1$.
- 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
- Check, by dividing by all primes. Remainder is 1 !!!
- We have obtained a contradiction.

In previous example, for "If P then Q", P is not really stated. Implicitly,

- P is laws of mathematics
- Q is statement that there is an infinite number of primes.

We assume $\neg Q$ and aim to obtain contradiction in law of mathematics.

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.

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 returns whether the program P will halt on input I.

Proof

- Assume that it is possible to write a program to solve the Halting Problem.
- Denote this program by $\textsc{HaltAnswerer}(\textsc{prog}, \textsc{inputs})$.
- $\textsc{HaltAnswerer}(\textsc{prog}, \textsc{inputs})$ will return
    - yes if prog will halt on inputs and
    - no otherwise.
- 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 $\textsc{HaltAnswerer}$ is effectively just working on two strings.

- 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.
- Our assumption that it is possible to write a program to solve the Halting problem has resulted in a contradiction (of fact that program cannot both halt and loop infinitely).

Principle of Induction

- 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 can be put in the form
- $P(n)$ is true for all integers $n \geq n_0$.
- Method:
  - 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 or for all integers less than or equal to n. This is called the inductive hypothesis.
  - Then show that the inductive hypothesis implies that $P(n+1)$ is also true.

Proving Correctness by Induction

Determine whether a number x is present in a sorted array $A[a..b]$

Algorithm 1 $\textsc{BinarySearch}(A, a, b, x)$

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

- To prove correctness, 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 $found = \textsc{BinarySearch}(A, a, b, x)$;
  Post: $found = x \in A[a..b]$ and A is unchanged
- The proof takes us from the precondition to the postcondition.
- **Base case:** $n = b - a + 1 = 0$
  - The array is empty, so $a = b + 1$
  - The test $a > b$ succeeds and the algorithm correctly returns false

- **Inductive step:** $n = b - a + 1 > 0$
  - **Inductive hypothesis:** Assume $\textsc{BinarySearch}(A, a, b, x)$ returns the correct value for all j such that $0 \leq j \leq n - 1$ where $j = b - a + 1$.
  - The algorithm first calculates $\textsc{mid} = \lfloor (a + b)/2 \rfloor$, thus $a \leq \textsc{mid} \leq b$.
  - If $x = A[\textsc{mid}]$, clearly $x \in A[a..b]$ and the algorithm correctly returns true.
  - If $x < A[\textsc{mid}]$, since A is sorted (by the precondition), x is in $A[a..b]$ if and only if it is in $A[a..\textsc{mid} - 1]$.
  - By the inductive hypothesis, $BinarySearch(A, a, \textsc{mid} - 1, x)$ will return the correct value since $0 \leq (mid - 1) - a + 1 \leq n - 1$.
  - The case $x > A[\textsc{mid}]$ is similar.
- We have shown that the postcondition holds if the precondition holds and BinarySearch is called.

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 $n/2$, with
- total time for doing the divide and combine steps $O(n)$.

Compute the asymptotic running time.

- Write *recurrence* for the running time:

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

- Guess that the running time is $T(n) \leq C_1 n^2$.
- Using $T(k) \leq C_1 k^2$ for $k < n$ as the induction hypothesis, we have:

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

- Cannot continue!

- Trick is to prove something stronger i.e. $T(n) \leq C_1 n^2 - C_2 n$.
- 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, find some n_0 where $T(n_0) \leq C_1 n_0^2 - C_2 n_0$.
- 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 means stronger induction hypothesis
  - easier to prove the inductive step
- But, useful only when the result you are trying to prove is true. ☺

Invariant Principle

- Invariants, 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 programs, requirements, etc.
- We will give an example of a loop invariant being used for proving the correctness of an iterative algorithm.

Summing an Array

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

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

- The key step in the proof is the invention of a condition called the loop invariant
    - Supposed to be true at the beginning of an iteration
    - 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

- In the example, we know that when the algorithm terminates with $i = b + 1$, the following condition must hold:sum $= \sum_{j=a}^{i-1} A[j]$.
- Use that as the loop invariant. Show that at the begining of the k-th loop, the loop invariant holds.
    - **Base Case:** $k = 1$
      Initialized to $i = a$ and sum $= 0$. Therefore $\sum_{j=a}^{i-1} A[j] = 0$.
    - **Inductive Hypothesis:** Assume sum $= \sum_{j=a}^{i-1} A[j]$ at the start of the loop's k-th execution.
      Let sum$'$ and i' be the values of the variables sum and i at the beginning of the $(k+1)$st iteration.
      In the k-th iteration, the variables were changed as follows:
      $\text{sum}' = \text{sum} + A[i]$
      $i' = i + 1$
      Using the inductive hypothesis we have

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

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

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

- Remains to show 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 $a \leq b + 1$.

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 $\textsc{PlaneColoring}(P, (l_1, \ldots, l_n))$

```
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
```

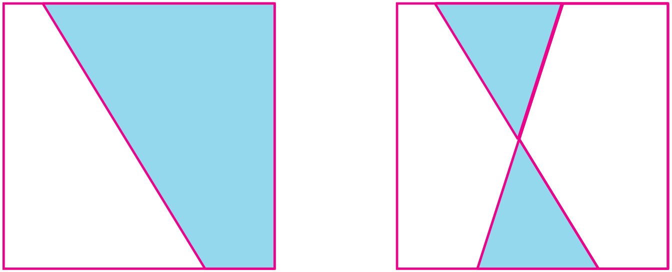


Figure: After the first and second lines are added.

- 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.

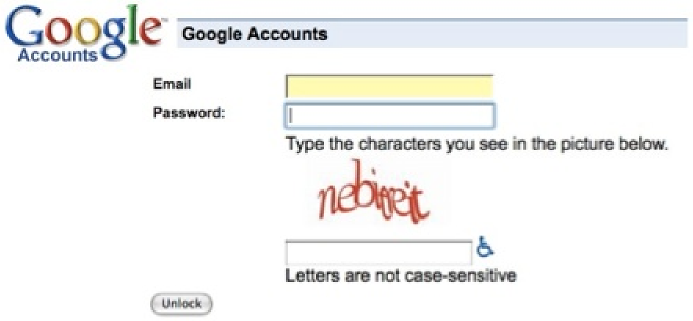
- **Loop invariant**: The plane P partitioned with lines $l_1, \ldots, l_{i-1}$ is a proper coloring i.e. any two regions that share a boundary have a different color.
- Base case: At first, the invariant is true as the plane is colored with one color.
- Assume that the invariant is true before the kth loop.
  - Within the loop, add the ith line.
  - Keep the colors on one side of the new line the same
  - Flip 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 - not changed by flipping all the colors on one side of the new line.
- Termination easy to see.

Reduction Principle

- A reduction is a transformation from one problem to another problem.
- Earlier, we used reduction to solve problems.
  - Transform a problem to one you know how to solve
  - Transform graph coloring to SAT, and use a SAT solver.
- Now, we use it to prove that one problem is as difficult as another problem.

Example: Completely Automated Public Turing test to tell Computers and Humans Apart (Captcha).

- How to ban robot web crawler from filling in forms and accessing websites that you would like to reserve for humans.
- Reduce form filling to difficult unsolved AI problem that humans can do!


Google Accounts

Email

Password:

Type the characters you see in the picture below.

Letters are not case-sensitive

NP-Completeness

- A decision problem is in the class NP (non-deterministic polynomial) if it is possible to verify the solution in polynomial time.
- A problem 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.
- 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.

Cook's Theorem

SAT is NP-complete.

Proof Idea:

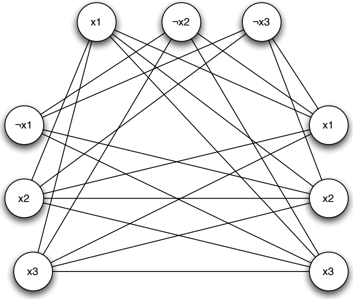
- By definition, it takes a polynomial number of steps to verify the solution.
- A computer is essentially a combinational circuit (using AND, OR and NOT gates).
- We can construct a circuit for verifying the solution for a problem in NP
  - Construct polynomial number of copies of the computer's circuit, one for each step
  - Output of one time step used as input of the next time step.
- Inputs to this circuit are the inputs to the verifying algorithm, includes the bits to the solution S that is being verified.
  - Whether there is input to the circuit that makes the output true, tells us the solution to the NP problem.
- Convert circuit into CNF.

- 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.
- We would like to show that the question of whether there is a clique of size k in a graph (CLIQUE) is NP-complete by reduction from 3-CNF-SAT.
- 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.

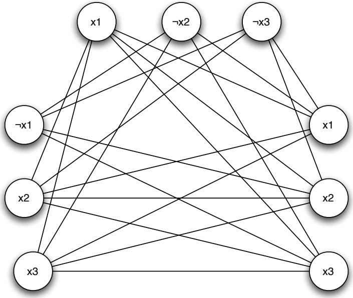
- For each clause C_i, create three nodes corresponding to the literals in the clauses. For example, the clause $C_i = (x \vee \neg y \vee z)$ would have three nodes v^i_x, $v^i_{\neg y}$ and v^i_z corresponding to the literals x, $\neg y$ and z within it.
    - Hence if we have m clauses, we will have $3m$ 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.



Graph derived from

$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_\vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

- Now show:
  - If SAT, then m-clique exists.
  - If UNSAT, then m-clique does not exist.
- 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.

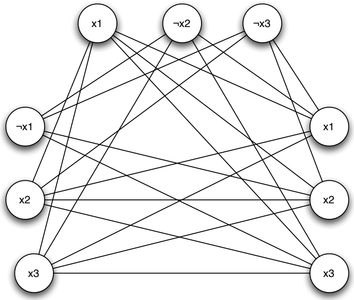


Graph derived from

$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_\vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

- 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.

- Conversely, show if the corresponding graph G has a clique of size m, the formula has a satisfying assignment (contrapositive of "If UNSAT, no clique").
  - 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.



Graph derived from

$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_{\vee} x_3) \wedge (x_1 \vee x_2 \vee x_3)$

Independent Set

- 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.

References

1. Antonella Cupillari, *The Nuts and Bolts of Proofs*, Academic Press.
2. Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithm*, MIT Press.