Tutorial 4

1 Greatest Common Divisor

In class this week, we studied invariants and how to use them. As another example of invariants, we will prove that Euclid's famous algorithm for computing the greatest common divisor of two numbers never gives the wrong answer. This is half of a proof that the algorithm is correct; the other half is to prove that the algorithm terminates. We do this by exhibiting a decreasing derived variable; a theorem presented in lecture then implies termination.

Definition 1 *The* greatest common divisor *(GCD) of two natural numbers x and y, not both zero, is the largest $n \in \mathcal{N}$ such that $n|x$ and $n|y$.*

For example, $GCD(6, 15) = 3$, $GCD(6, 7) = 1$, and $GCD(2, 0) = 2$. $GCD(0, 0)$ is undefined. Computing greatest common divisors is important in cryptography.

One way to compute the GCD of two numbers is to factor both into primes and then pull out prime factors shared by both. For example:

$$\begin{aligned} 24 &= 2^3 \cdot 3 \\ 36 &= 2^2 \cdot 3^2 \\ &\Rightarrow GCD = 2^2 \cdot 3 = 12 \end{aligned}$$

The problem is that factoring is very slow. Fortunately, the Greek mathematician Euclid thought of a better algorithm. (He apparently never implemented this algorithm; perhaps he was too lazy to learn C.)

The input to procedure `Euclid` is a pair of numbers (x, y). There are only two steps in the algorithm:

1. If $y = 0$, return x.

2. Otherwise, return `Euclid`$(y, x \bmod y)$.

(Recall that $x \bmod y$ denotes the remainder when x is divided by y.)

As an example, suppose that the procedure is called with inputs $x = 22$ and $y = 16$. This would generate a sequence of recursive calls: `Euclid`(22, 16), `Euclid`(16, 6), `Euclid`(6, 4), `Euclid`(4, 2), `Euclid`(2, 0). The final call returns 2. The value 2 propagates all the way back up the recursion and is output as the final answer. Sure enough, $GCD(22, 16) = 2$.

Here is the theorem that we want to prove about Euclid's algorithm:

Theorem 1 *If* `Euclid`(x, y) *terminates, then it returns $GCD(x, y)$.*

We can use the state machine/invariant approach to prove this theorem. The first problem is to define the three components of the state machine, Q, Q_0, and δ:

$$\begin{aligned} Q &= \{(a, b) \mid a, b \in \mathcal{R}\} \\ Q_0 &= \{(x, y)\} \\ \delta &= \{(a, b) \rightarrow (b, a \bmod b) \mid b \neq 0\} \end{aligned}$$

Two aspects of these definitions might be disturbing. First, the start state contains variables, x and y. This will not be a problem; we will treat x and y as constants in our analysis. The upshot is that we are defining a somewhat different state machine for each input.

Second, we have not defined a transition for the case when $b = 0$, even though this corresponds to step 1 of the algorithm. This is the termination condition, which we are deferring.

The only remaining preparatory work before the proof is choosing a predicate. Looking back at the example execution suggests the predicate $GCD(a, b) = GCD(x, y)$. That is, any stage the GCD of the remaining numbers is equal to the GCD of the original two numbers. Now we are ready to prove the theorem.

Proof. The proof uses the Invariant Theorem.

Let $P((a, b))$ be the predicate that $GCD(a, b) = GCD(x, y)$.

First, we must show that the predicate holds for all start states. This is trivial because the only start state is (x, y) and $GCD(x, y) = GCD(x, y)$.

Next, for each transition $(a, b) \rightarrow (a', b') \in \delta$, assume $GCD(a, b) = GCD(x, y)$ to prove $GCD(a', b') = GCD(x, y)$. It is sufficient to show that $GCD(a', b') = GCD(a, b)$. Futhermore, since all transitions in δ are of the form $(a, b) \rightarrow (b, a \bmod b)$, we have only to check that $GCD(b, a \bmod b) = GCD(a, b)$.

Let $s = GCD(a, b)$ and $t = GCD(b, a \bmod b)$. We will prove that $s = t$ by showing that each divides the other.

First, we show that s divides t. By definition of GCD, $s \mid a$ and $s \mid b$. For some integer c, $a \bmod b = a - bc$, and so $s \mid a \bmod b$. Since $s \mid b$ and $s \mid a \bmod b$, $s \mid t$.

Now, we show that t divides s. Again, by definition of GCD, $t \mid b$ and $t \mid (a - bc)$. These facts imply $t \mid a$, and so $t \mid GCD(a, b)$ and $t \mid s$. ■

This proves that at every stage the algorithm stores two numbers whose GCD is the same as the original input pair. If the second number ever becomes zero, then the correct answer will be returned by step 1 of the algorithm.

Now we address the question of whether the algorithm terminates, and if so how quickly it terminates. Recall from lecture the theorem about decreasing derived variables.

Theorem 2 *If $f : Q \rightarrow \mathcal{N}$ is a decreasing derived variable then the length of any execution starting at q_0 is at most $f(q_0)$.*

We will now use this to show that Euclid's algorithm terminates in at most x steps.

First let us establish a suitable invariant.

Claim 1 *If $x > y$ then $P((a, b)) : a > b$ is an invariant.*

Proof. To prove that P is an invariant, we must show that it hold for all start states and is preserved by all transitions. The invariant holds trivially in the start state, since by assumption $x > y$.

Next, we have to show that for each transition $(a, b) \rightarrow (a', b') \in \delta$, $P(a, b) \Rightarrow P(a', b')$. That is, we asum $a > b$ to prove $a' > b'$, where the algorithm specifies that $a' = b$ and $b' = a \bmod b$. This holds because $a \bmod b < b$ by definition of the mod operator. ∎

Let f be the derived variable $f((a, b)) = a$.

Claim 2 *If $x > y$ then f is a decreasing derived variable.*

Proof. Consider a transition from (a, b) to $(b, a \bmod b)$. In the state (a, b) we know from the invariant that $a > b$.

Therefore $f((a, b)) = a > b = f((b, a \bmod b))$. ∎

From the the second claim and the theorem proven in lecture, we conclude that if we start in a state where $x > y$ then the algorithm terminates in at most $f((x, y)) = x$ steps.

2 Transitive Closure

Given a directed graph $G = (V, E)$, the *transitive closure* of G is defined as the directed graph $G' = (V, E')$ where $E' = \{(u, v) \mid \text{There is a directed path from } u \text{ to } v \text{ in } G\}$. If E' also contains self-loops (u, u) for all vertices u, then G' is called the *reflexive-transitive closure* of G.

The transitive closure is a useful notion in many settings. For example, given a set of precedence constraints, the transitive closure of the corresponding graph gives us the set of all constraints that are implied.

How can we compute the transitive closure? Here is a very simple algorithm.

1. Insert a loop at every node if one is not already present.
2. Repeat:
   If (u, x) and (x, v) are edges in the current graph,
   and (u, v) is not an edge, then add (u, v) to the graph.
   If there are no such $(u, x), (x, v)$ then stop.

Does the algorithm terminate? Does it give us the transitive closure when it terminates? How many steps does it take ? We will answer these questions using the tools developed in lecture, namely invariants and derived variables.

First we set up a state machine for transitive closure. Each state is a directed graph $G = (V, E)$. The start state G_0 is the graph whose transitive closure we wish to compute (we will treat G_0 as a "constant"). There is a transition from a state $G_1 = (V, E_1)$ to a state $G_2 = (V, E_2)$ iff there is some pair of edges $(u, x), (x, v)$ in E_1 and (u, v) is not in E_1, and $E_2 = E_1 \cup \{(u, v)\}$. Notice that this state machine is *nondeterministic*.

Let $C(G)$ be the set of pairs (u, v) such that there is a path from u to v in G.

Let $P(G)$ be the predicate: $C(G) = C(G_0)$.

Claim 3 *$P(G)$ is an invariant.*

Proof. First we observe that $P(G_0) : C(G_0) = C(G_0)$ is true. Then consider a transition G_1 to G_2. So there are edges $(u, x), (x, v) \in E_1$, $(u, v) \notin E_1$ and $E_2 = E_1 \cup (u, v)$.

Assume $P(G_1)$ is true, so $C(G_1) = C(G_0)$.

We have to verify $P(G_2)$. Suppose for some pair of vertices y, z, there is a path from y to z in G_1. Then it continues to be a path in G_2 (we only added an edge). Hence $C(G_1) \subseteq C(G_2)$. Now suppose for some pair of vertices y, z there is a path in G_2 from y to z. If this path does not use the edge (u, v) then it is a path in G_1 also. If it uses (u, v) then there is another path from y to z that uses (u, x) and (x, v) instead of (u, v) and it is present in both G_2 and G_1. Hence $C(G_2) \subseteq C(G_1)$. So we conclude that $C(G_2) = C(G_1) = C(G_0)$. ■

To show termination and to derive a bound on the number of steps, we use the derived variable $f : Q \to \mathcal{N}$ defined as

$f(G) = |E|$. (The number of edges in G.)

Claim 4 *f is an increasing derived variable.*

Proof. At each transition we add an edge (u, v) that is not already present. There are no edges deleted. So for a transition from G_1 to G_2, $f(G_2) = f(G_1) + 1$. ■

Claim 5 *The algorithm terminates in at most n^2 steps for a graph with n vertices.*

Proof. $f(G_0) \geq 0$

$f(G) \leq n^2$ as no graph has more than n^2 edges.

Since f is increasing, it follows that the algorithm terminates in at most n^2 steps. ■

This is a variation on the proof method outlined in class. There we used a *decreasing* derived variable. Here we use the same idea, but it is convenient to have an increasing derived variable.

Finally let us verify that it gives us the right answer when it terminates.

Claim 6 *The algorithm stops at a state $G' = (V, E')$ corresponding to the transitive closure of the start state G_0.*

Proof. We need to show that for any pair u, v

$(u, v) \in E'$ iff there is a path from u to v in G_0.

If $(u, v) \in E'$, then $(u, v) \in C(G')$. Using the fact that $P(G)$ is an invariant, $C(G') = C(G_0)$, hence $(u, v) \in C(G_0)$ and there is a path from u to v in G_0.

To prove the reverse implication, suppose for a contradiction that there is a path from u to v in G_0 and $(u, v) \notin G'$. Then there is also a path from u to v in G'. Let $p = (u, u_1, \ldots, u_k = v)$ be the shortest path from u to v in G'. The edge (u, u_2) is not in G', because if it is then $(u, u_2, \ldots, v)$ is a shorter path. Since (u, u_1) and (u_1, u_2) are edges in G', there is a transition out of G' to the graph obtained by adding the edge (u, u_2). This is a contradiction as we assumed that G' is a terminal state. ■

There is another way to compute the transitive closure of a graph. Let us assume we are given the adjacency matrix A of the graph. Then here is the algorithm for reflexive transitive closure:

1. Add in self-loops, i.e. make diagonal entries equal to 1 to get a new matrix B.
2. Compute B^n
3. Replace all nonzero entries by 1, keep 0 entries.

This algorithm is convenient because B^n can be computed in about $\log n$ steps by repeatedly squaring the matrix B. For example, to compute B^8.

Theorem 3 *Let G be a directed graph with vertices $v_1, v_2, \ldots v_n$. Let B be the adjacency matrix of G augmented with self-loops. Then B^n_{ij} is non-negative if and only if there is a path from v_i to v_j in G.*

Proof. The proof is by induction.

Let $P(n)$ be the predicate that B^n_{ij} is non-negative if an only if there is a path from v_i to v_j in G of length at most n.

In this base, $P(1)$ holds because there is a path of length 1 between v_i and v_j iff there is an edge between these two vertices. Similarly, there is a path of length 0 from a vertex to itself.

In the inductive case, assume $P(n)$ to prove $P(n+1)$. Consider the matrix product $B^n B = B^{n+1}$. By definition of matrix multiplication, $B^{n+1}_i j$ is non-negative if and only if for some k, B^n_{ik} is nonzero and B_{kj} is nonzero. This implies that there is a path from v_i to v_k of length at most n and a path from v_k to v_j of length at most 1. As a result, there is a path of length at most $n+1$ from v_i to v_j. ■

3 Team Problems

1. *Efficiency of Euclid's algorithm.*

   (a) Show that Euclid's GCD algorithm uses at most $2\log_2 x$ modulo operations. [Use derived variables]

   (b) Let F_n denote the n^{th} Fibonacci number. Show that:

      i. If $2 \leq x \leq F_n$ then the algorithm uses at most $n-2$ modulo operations.

      ii. For $n \geq 2$, $F_n \geq (\frac{3}{2})^{n-2}$.

      iii. Using the previous two results show that Euclid's algorithm actually uses at most $1.71\log_2 x$ modulo operations if $x \geq 2$.

2. *Choosy mating.*

   (a) We have the mating problem again, but now both boys and girls are more choosy about their partners. That is, among a set of n boys and n girls, each boy is unwilling to drop below his k^{th} choice and each girl is unwilliling to below her m^{th} choice. So it is possible that some of them are not married. How many pairs can we be sure of having in general ?

   (b) What if only the boys were choosy, i.e., the girls are willing to marry anyone (they still have a ranked list of preferences), but the boys will not go below their k^{th} choice ? How many pairs will the algorithm from class (TMA) assure us ?