CSEE E6861y **Overview of Fast Prime Generation** Handout 17
Prof. Steven Nowick February 18, 2016

This handout presents an overview of a fast algorithm for **recursive prime generation**.

Introduction

As with tautology checking and complementation, the fast prime generation algorithm uses a recursive "divide-and-conquer" approach. Given an initial cover, the algorithm recursively divides the cover into smaller pieces until termination conditions are met. Results are then returned and reassembled into a final solution. For prime generation, this solution is the set of all prime implicants of the initial function.

An important aspect of the algorithm, as with the previous recursive algorithms, is that *properties of unate functions are exploited,* to simplify the recursion. Therefore, this algorithm is another application of the "unate recursive paradigm".

The algorithm below is the one currently used in *espresso-exact* to generate all prime implicants. It is described in Rudell's thesis, ch. 2.5.1 (Handout #8). However, Rudell's description is actually for more general "multi-valued" Boolean algebras so some of his notation may not be clear. This handout describes the same algorithm, but restricted to the more usual "binary-valued" (0/1) Boolean algebra.

An interesting historical note: The original *espresso-exact* program used a different prime generation algorithm, described in Rudell ch. 2.5.2. However, that algorithm was replaced by the consensus algorithm below (also, ch. 2.5.1) around 1987-88. The motivation for the change is described in Rudell ch. 2.5.3: this new algorithm has much better runtime.

The Prime Generation Problem:

Given a cover F, corresponding to the *union of the ON-set and DC-set* of a Boolean function f, return the set of all prime implicants of f.

Definitions.

First, some basic definitions, which will be used in the formulation of the recursive prime generation algorithm.

Given products p_1 and p_2, the two products **differ in a variable x** if p_1 has literal x and p_2 has literal $\overline{x}$ (or vice-versa). If both products have literal x, or both have literal $\overline{x}$, or if variable x (complemented or uncomplemented) does not appear in at least one of the products, then the two products **do not differ in variable x**. For example, if $p_1 = abc'$ and $p_2 = acd'$, then (i) the two products differ in variable c, and (ii) the two products do not differ in variables a, b and d.

The **distance between two products** is the number of inputs where they differ. Intuitively, the distance between two products indicates how far apart they are in a hypercube (at their closest points). In the above example, p_1 and p_2 differ in one variable, so they have distance 1. If $p_1 = ab'c$ and $p_2 = a'b'c'd$, then the two products differ in two variables (a and c), so their distance is 2.

If two products have distance 1 or greater, then they **do not intersect**, *i.e.,* they have an empty intersection. If two products have distance 0, then they **intersect**. In this case, the **intersecting cube,** q, is formed in the usual way: If literal x appears in either p_1 or p_2, then x appears in q. If literal $\overline{x}$ appears in either p_1 or p_2, then $\overline{x}$ appears in q. If neither x nor $\overline{x}$ appears in either of the products, then it does not appear in q, either. For example, the intersection of products abc' and bd is: $abc' \cap bd = abc'd$. Note that variable d does not appear in p_1, but literal d appears in p_2, so the literal d appears in the intersection.

The **consensus** of two products will be defined if they *do not intersect*.[1] If two products have distance 2 or greater, their consensus is empty (*i.e.* no consensus). Otherwise, if two products have distance 1 (that is, differ

[1] Actually, there is a special definition of consensus when two products intersect, but we will not use this.

in one variable), their consensus is, informally, the product which can be formed to *"connect them"*. More formally, suppose p_1 and p_2 have distance 1, and differ in variable x. Suppose literal x appears in p_1 and $\overline{x}$ appears in p_2, so the two products do not intersect. The consensus of p_1 and p_2 is formed by (i) deleting $x/\overline{x}$ from the two products, then (ii) forming the intersection of the resulting products.

Example. As an example of consensus, suppose $p_1 = ab'c'$ and $p_2 = bc'd'$. Clearly, p_1 and p_2 do not intersect, since they differ in variable b. Since they only differ in variable b, p_1 and p_2 have distance 1. To form the consensus of p_1 and p_2, first (i) delete the b variable from both, to get ac' and $c'd'$, respectively. Next, (ii) intersect these two new cubes, to get: $ac' \cap c'd' = ac'd'$. Intuitively, cube $ac'd'$ "connects" p_1 and p_2, in the region where the two cubes are "adjacent" (*i.e.,* in $ab'c'd'$ in p_1 and in $abc'd'$ in p_2).

The notion of consensus can be generalized from a pair of products to two sets of products. The **consensus of two sets of products**, A and B, written $consensus(A, B)$, is simply formed by taking the consensus of each pair of products, one cube from A and one cube from B. That is, it is the "pairwise" consensus of the two sets A and B. The set of all resulting products is the consensus of sets A and B.

The Fast Prime Generation Algorithm

We now describe a fast recursive algorithm for prime generation.

Overview

Fast prime generation is based on simple idea. Given a function f and a splitting variable x, *each prime implicant of f falls into one of 3 categories:* either (i) it contains literal x, (ii) it contains literal $\overline{x}$, or (iii) it does not contain x or $\overline{x}$. Category (i) contains primes in one half of the hypercube of the function (where $x = 1$). Category (ii) contains primes in the other half of the hypercube (where $x = 0$). Finally, Category (iii) contains primes which span both halves of the hypercube.

As an example, given the Boolean function f with cover $F = xyz' + x'y' + y'z$, then xyz' is in Category (i), $x'y'$ is in Category (ii), and $y'z$ is in Category (iii) (x does not appear in this prime).

Category (i) and (ii) primes can be generated recursively, by finding the primes of two respective cofactors: F_x and $F_{x'}$ (to be described shortly). The key interesting issue is: how to find the remaining primes (Category (iii)), which span both halves of the hypercube?

The following fundamental theorem provides the answer:

Prime Consensus Theorem. Let f be a Boolean function, with cover F (which covers the entire ON-set and DC-set). Also, let x be any input variable. Then the prime implicants of f can be partitioned into 3 sets:

(i) $P1$, which contains primes having literal x;
(ii) $P0$, which contains primes having literal $\overline{x}$;
(iii) $P-$, which contains primes with no x or $\overline{x}$. Each prime p in $P-$ is the *consensus* of two other primes, p_1 and p_0, where p_1 is from $P1$ and p_0 is from $P0$. That is, $p = consensus(p_1, p_0)$, for some prime p_1 in $P1$ and p_0 in $P0$.

This key theorem states that every prime in category (iii) can be formed as the *consensus* of some prime in $P1$ and some prime in $P0$. Intuitively, such a consensus cube is the shortest "connecting cube" between a pair of cubes which are distance-1 apart. Thus, if you *already* have sets $P1$ and $P0$, *all remaining primes* (*i.e.,* those in $P-$) *can be simply generated as follows:*

> For *each* pair of primes, (p_1, p_0), where p_1 is in $P1$ and p_0 is in $P0$, generate the consensus cube $p_{1/0} = consensus(p_1, p_0)$. The resulting set $P1/0$ of all such consensus cubes contains all the primes of $P-$.

More formally, this is notated as the consensus of the two sets: $P1/0 = consensus(P1, P0)$.

Finally, one small adjustment must be made: although set $P1/0$ contains all prime implicants of $P-$, $P1/0$ *may also contain some smaller non-primes!* **These non-primes must be deleted.** To do this we introduce a new clean-up operator, which deletes smaller cubes:

Definition: Single-Cube Containment (SCC). Given any two cubes, $c1$ and $c2$, cube $c1$ is **single-cube contained** in $c2$, if $c2$ contains (i.e. covers) $c1$. For example, if $c1 = wy'$ and $c2 = wxy'z$, then $c2$ is single-cube contained in $c1$. This definition can be extended to a *set of cubes*: given a set C of cubes, $\{c1, c2, \ldots, cn\}$, then the **single-cube containment of set C** is the resulting set after applying single-cube containment to *every* pair of cubes. For example, if $C = \{abc, ab', bcd, ab'd, a'd, a'bcd\}$, then the single-cube containment of C (written *SCC(C)*), is the set {abc, ab', bcd, a'd}.

Thus, *SCC* is a "clean-up operator," which takes a set of cubes and deletes each cube which is completely contained in another cube in the set. Using *SCC*, we can now formally state how to generate $P-$, if we are given sets $P1$ and $P0$:

$$P- = SCC(consensus(P1, P0))$$

That is, the set $P-$ is formed by taking the pairwise consensus of all cubes in $P1$ and $P0$, then deleting any resulting smaller (non-prime) cubes contained inside other cubes.

Generating P1 and P0.

The above result indicates how to generate $P-$, if you are given $P1$ and $P0$. Next, we address how to generate the original sets $P1$ and $P0$. Set $P1$ is the set of all primes which have literal x. To generate $P1$, we first generate all primes of a cofactor of cover F: $Primes(F_x)$. Once these primes are generated, each result is ANDed with literal x. The resulting set contains all primes of $P1$. However, as before, *the set may also contain some non-primes,* which will be contained in $P-$ consensus cubes. These will be deleted later.

Similarly, set $P0$ is the set of all primes which have literal x'. To generate $P0$, we first generate all primes of a cofactor of cover F: $Primes(F_{x'})$. Once these primes are generated, each result is ANDed with literal x'. The resulting set contains all primes of $P0$. Again, the set may also contain some non-primes, which will be contained in $P-$ consensus cubes. These will be deleted later.

Fast Prime Generation: Algorithm.

Finally, we now describe the complete prime algorithm used in *espresso-exact*. The algorithm generates the set of primes, by producing the 3 sets: $P1$, $P0$ and $P-$. First, it recursively generates primes of the two halves of the domain, F_x and $F_{x'}$, and then AND's them with corresponding literals (x and x', respectively). Once these are generated, it then uses the Prime Consensus Theorem to produce the set of all consensus cubes, $P-$, spanning between pairs of cubes from the two halves. Finally, it removes all non-primes. The result is the complete set of prime implicants of the original function:

Recursive Prime Generation Theorem. Let f be a Boolean function, with cover F (which covers the entire ON-set and DC-set). Also, let x be any input variable. Then the prime implicants of f can be generated as follows:

$$Primes(f) = SCC(A1 \cup A0 \cup consensus(A1, A0)),$$

where $A1 = x \cdot Primes(F_x)$ and $A0 = x' \cdot Primes(F_{x'})$.

This key theorem formally summarizes how the primes of f can be generated recursively, and effectively is an outline of the algorithm. The problem of finding primes of f is transformed into the problem of finding the primes of two simpler cofactors, f_x and $f_{x'}$. These are used to generate $A1$ and $A0$. $A1$ contains all the primes of $P1$, but may also contain non-primes which are contained in cubes of $P-$. $A0$ contains all the primes of $P0$, but may also contain non-primes which are contained in cubes of $P-$. Once these two results are returned, a third set is then generated from them, $consensus(A1, A0)$, containing cubes connecting each distinct pair of cubes from the two sets (i.e. connecting cubes with no x variable). All non-primes are deleted, using the SCC operator. The resulting set contains precisely all the primes of F.

The declarative theorem corresponds directly to an algorithm (i.e. procedure), as follows.

Step #1. Check if F satisfies termination conditions (B1-B4 and U1 below). If so, immediately return the result, $Primes(F)$. The algorithm is done.

Step #2. If the algorithm is not done, and none of the termination conditions applies, then the cover is recursively split in two halves, and the same algorithm is now repeated on each half. A splitting (or branching) variable, x, is selected. Recursively compute two sets: (i) $A1 = x \cdot Primes(F_x)$, and (ii) $A0 = x' \cdot Primes(F_{x'})$. $A1$ contains the primes of $P1$ (and possibly some non-primes). $A0$ contains the primes of $P0$ (and possibly some non-primes). Finally, the two resulting sets, $A1$ and $A0$, are used to create a third set, $consensus(A1, A0)$, containing cubes which span between each pair of cubes of these two sets. After deleting non-primes (applying the *SCC* operator), the union of the 3 sets is precisely the set of all primes of f.

Choice of Splitting Variable. The choice of splitting is important. The same strategy can be used, as is described in Handout #11: *Overview of Tautology Checking.*

Basic Rules for Termination

Basic rules are used to terminate recursion, as in tautology checking and complementation. As in complementation (and unlike tautology-checking), an actual cover is returned: the set of all primes of the given function.

B1. **Cover F includes the Universal Cube.** Here, F is a tautology. Return a cover containing the universal cube.

B2. **Single-Input Dependence.** If the function depends on only one input x (*i.e.,* all other input columns contain only '-'), and the x column contains both 1's and 0's, then the function is a tautology. Return a cover containing the universal cube.

B3. **Cover F is Empty.** A cover is *empty* if it contains no cube. In this case, the function is all 0. Return the empty cover (contains no cubes).

B4. **Cover F contains a Single Cube.** In this case, the cube itself must be the only prime of the function. Return F (which contains the single cube).

Advanced Rules for Termination: Unateness Conditions

A powerful advanced rule can be applied, to terminate early, when a cover is unate. This rule can be used even if B1-B4 do not apply. The following theorem, presented by Robert Brayton *et al.*, describes the *unate termination condition:*

Unate Prime Theorem. Let f be a Boolean function, with cover F (which covers the entire ON-set and DC-set). *If cover F is unate, then it contains all of the prime implicants of function f.* In addition, F may contain some non-primes, which can be deleted. That is:

$$Primes(f) = SCC(F).$$

This theorem presents a remarkable and comprehensive result: if F is any unate cover, then it must contain all the prime implicants of function f! In addition, F may also include some non-primes. **No further recursion is necessary.** Once the non-primes are deleted using *SCC* operator, the resulting cover is precisely the set of *all prime implicants* of f. Thus, if F is unate, we simply terminate and delete all non-primes, using the *SCC* operator. The result is *exactly* the set of all prime implicants of F. The resulting cover is simply returned: $SCC(F)$.

It is beyond the scope of the course to prove this theorem, but some intuition is presented in the class lectures. This theorem can be formalized into a termination rule:

U1. **Unate Cover.** If the cover F is unate, then terminate. Return $SCC(F)$, which is precisely the set of all primes of F.