Алгоритмы для управления Бинарными функциями основанные на графах.

Рандал Браянт

Краткий обзор: в этой статья мы представляем новую структуру данных для представления Бинарных функций и связанное множество манипуляционных алгоритмов. Функции представлены ориентированными, нецикличными графами способом похожим на представление представленное Лии (1) и Акерсом (2), но с добавленными ограничениями на порядок искомых переменных в графе. Несмотря на то что функциональная сложность, в худшем случае экспоненциально зависит от числа аргументов функции, большинство встречающихся функций имеют довольно стоящее представление. Наши алгоритмы имеют временную сложность пропорциональную размеру графа, с которым ведется работа, и эффективны до тех пор пока граф не станет слишком большим. Мы представляем результат экспериментов применения этих алгоритмов к проблемам логических вычислений, которые доказывают практичность нашего представления.

Ключевые слова – Бинарная функция, бинарная схема решений, верификация логических схем, символьная манипуляция.

Введение

Бинарная алгебра представляет собой краеугольный камень компьютерной науки и разработки цифровых систем. Большинство проблем в разработке и тестировании цифровой логики, искусственного интеллекта, и комбинаторики могут быть выражены как последовательность операций с бинарными функциями. Использование этих методов является эффективным алгоритмом для представления и манипуляции бинарными функциями символически. К сожалению, многие задачи, которые хотелось бы представить в виде бинарной функции, являются NP-сложными задачами. Следовательно, все известные подходы для представления этих операций потребуют, в худшем случае, машинного времени, величина которого растет экспоненциально в зависимости от размера задачи. В этом и состоит сложность оценки эффективности различных методов представления и манипуляции Бинарных функций. В худшем случае, все известные подходы представления также малоэффективны как и простой подход представления функции с помощью таблиц истинности. На практике, использование более эффективные представления и манипуляционные алгоритмы, мы можем избежать экспоненциальных расчетов.

Множество методов были разработаны для представления и манипуляции бинарными функциями. Основанные на классическом представлении с помощью таблиц истинности: карты Карно, или каноническая форма sum-of-product являются довольно непрактичными, т.к. каждая функция n аргументов требует размерность представления 2^n или более. Более практично использовать представления которые, по крайней мере для многих функций, не имеют экспоненциальную сложность. Пример такого представления включает в себя пониженную сумму продуктов (элементов) (или эквивалентную множеству простых кубов) и приведен к виду однородной функции. Но у этих методов есть и ряд недостатков. Во-первых, остаются все же функции которые требуют экспоненциальную сложность представления. Например, функции «четность» и «нечетность» являются худшим примером во всех этих представлениях. Второе, в то время как некоторые функции имеют смысл в таком представлении, производя простые операции такие как комплементация могут произвести функцию с экспоненциальной сложностью представления. В конце концов, ни одно из этих представлений не дает каноническую форму, т.е., исходная функция может иметь несколько представлений. Следовательно, тестирования на эквивалентность и выполнимость могут быть довольно трудоемкими.

В результате этих характеристик, большинство программ работающие с последовательностью операций с бинарными функциями имеют скорее хаотичное поведение. Они движутся с нормальным шагом, но в конце концов «взрываются», толи выходят за пределы области расчета или невозможность продолжения просчета с соизмеримыми затратами времени.

В этой статье мы представляем новый класс алгоритмов для манипуляции с бинарными функциями представленными в виде направленного нецикличного графа. Это преставление похоже на бинарную диаграмму решений представленную Лии и в дальнейшем описанной Акерсом. Тем не менее, мы положили дальнейшее ограничение на расположение результативных переменных – вертикально. Эти ограничения позволяют более эффективно разрабатывать алгоритмы манипуляции представлений.

Наши представления имеют ряд преимуществ по сравнению с предыдущими подходами манипуляции с бинарными функциями. Во-первых, большинство общих встречающихся функций имеют приемлемое представление. Например, все симметричные функции (включая «четность» и «нечетность») представлены графами где количество узлов равно квадрату количества аргументов. Во-вторых, представление программы основанной на наших алгоритмах, когда обработка последовательности операций ухудшается медленно, если вообще. Временная сложность любой простой операции ограничена размером графа для функции с которой мы работаем. Например, вычисление функции требует время пропорциональное размеру графа, в то время как комбинирование двух бинарных функций (такие как конъюнкция, вычитание, и импликация – это отдельные случаи) требует времени пропорционально двум размерам графа. Наконец, наше представление является канонической формой упрощенного графа, т.е. каждая функция имеет уникальное представление. Отсюда, проверка на эквивалентность сводится к сравнению на идентичность двух графов, а проверка удовлетворения сводится к проверке функции константы.

К сожалению наше представление не имеет своего множества нежелательных характеристик. Прежде всего мы должны определится с порядком единых входных параметров в качестве аргументов для всех представляемых

функций. Для некоторых функций размер графа представляющего функцию очень чувствителен к этому порядку. Проблема вычисления порядка минимизации графа является NP-сложной задачей. Наш опыт, тем не менее, показывает, что человек понимающий проблему, может выбрать подходящий метод без такой большой сложности. Похоже что использование маленького множества эвристик может привести к тому, что программа сама сможет выбрать адекватный метод. Более того, есть несколько функций, которые могут быть представлены бинарными выражениями или логическими сетями разумного размера, но для всех входных последовательностей представление в виде функционального графа слишком большое для использование на практике. Например, мы доказали в приложении, что функции описывающие результат целочисленного умножения представляются графом, размер которого растет экспоненциально, независимо от входных последовательностей. Если не рассматривать целочисленное умножение, наш опыт показывает, что такие функции редко появляются в приложениях. Для других классов проблем, особенно в комбинаторике, наши методы практичны, но только в ограниченных условиях.
Многообразие графического представления дискретных функций в настоящий момент интенсивно исследуется. Ряд литературы по этому разделы представлен сайтами Морета, содержащими более 100 ссылок, но ни одна не описывает алгоритм для составления программы управления бинарными функциями. К счастью, Гопкрофт и Шмидт изучили свойства графов используя ограничения похожие на наши, и показали что два графа могут быть проверены на функциональную эквивалентность за полиномиальное время, и что некоторые функции потребуют граф большего размера при таких ограничениях, чем при более мягких ограничениях. Пейн описал технику похожую на нашу для сокращения размера графа представляющего некоторую функцию. Наши алгоритмы применения операции к двум функциям, и для сравнения двух функций являются новыми, тем не менее эта возможность приведена к программе символической манипуляции.
Следующая часть этой статьи содержит формальное представление функциональных графов. Мы определили графы, функции, которые они выражают, и «класс» сокращенных графов. Затем мы доказали ключевое свойство сокращенных функциональных графов: то что они являются каноническим представлением бинарной функции. Далее мы рассмотрели ряд примеров, и обсудили эффективность нашего представления. Исходя из этого мы определили множество алгоритмов для манипуляции с бинарными функциями представленными в виде графов. Эти алгоритмы включают в себя основную технику алгоритмов для работы с графами.

Примечание

Мы приняли, что все функции имеют количество аргументов равное n. Выражая такую систему как логическая сеть комбинаторики или бинарное выражение как бинарную функцию, нам необходимо определиться с порядком входных данных, и этот порядок должен быть обязателен для всех функций.
Замена некоторого аргумента x_i функции f константой b, называется «ограничение f» и записывается как:
$f|_{xi=b}$, для любого аргумента $x_1,\ldots,x_n$,
$f|_{xi=b}(x_1,\ldots,x_n)=f(x_1, \ldots , x_{i-1} , b , x_{i+1} , \ldots ,x_n)$
Замена некоторого аргумента x_i функции f функцией g, называется «композиция f и g» и записывается как:
$f|_{xi=g}$, для любого аргумента $x_1,\ldots,x_n$,
$f|_{xi=g}(x_1,\ldots,x_n)=f(x_1, \ldots , x_{i-1} , g(x_1,\ldots,x_n) , x_{i+1} , \ldots ,x_n)$
Некоторые функции не зависят от всех аргументов. «Множество зависимости» функции f, обозначается I_f, и одержит все те аргументы от которых данная функция зависит, т.е.
$I_f=\{i|f|_{xi=0}<>f|_{xi=1}\}$

Функции, которые при любом значении входного параметра дают результат 1 (0), называются «константа 1» («константа 0»). Обе эти функции имеют пустое множество зависимости.
Бинарная функция может быть рассмотрена некоторое подмножество n-мерного бинарного множества, на котором функция дает результат 1. «Множество удовлетворения» обозначается S_f и записывается как
$S_f=\{(x_1,\ldots,x_n)|f(x_1,\ldots,x_n)=1\}$

II. Представление

В этом разделе мы описали графическое представление бинарных функций и доказали, что они имеют канонических вид.
Определение 1: функциональный граф имеет корень, и является направленным графом с множеством вершин V, содержащих два значения. Нетерминальная вершина v имеет атрибуты и индекс аргумента index(v)Є{1,…,n} и две ветви low(v) и high(v) Є V. Терминальная вершина v имеет значение атрибута value(v)Є{0,1}.
Более того, для любой нетерминальной вершины v, если ее low(v) также является нетерминальной вершиной, то мы имеем index(v) < index(low(v)). Таким же образом, если high(v) нетерминальная вершина, то index(v) < index(high(v)).
Исходя из нашего определения, функциональный граф формирует подмножество условной бинарной диаграммы принятия решения. Также функциональный граф должен быть ацикличным.
Теперь определим связь между функциональным графом и бинарными функциями.
Определение 2. Функциональный граф G с корневой вершиной v рекурсивно описывает функцию fv как:

1) если v терминальная вершина

а) if value(v) = 1, then fv = 1

б) if value(v) = 0, then fv = 0

2) Если v нетерминальная вершина с index(v) = i, тогда fv – функция

fv(x1,…,xn) = (-)xi*flow(v)(x1,…,xn) + xi*fhigh(v)(x1,…,xn)

Другими словами, мы можем рассматривать множество значений аргумента x как множество путей графа, начиная с его корня, где есть некоторое значение и далее путь продолжается в «младшее поддерево» если значение аргумента 0 или в «старшее поддерево» - если аргумент равен 1. Значение же самой функции для данной последовательности аргументов будет значением терминальной вершины в конце графа. Важно отметить, что пути отмечены множеством уникальных значений. И каждая вершина имеет по крайней мере один путь, т.е. нет «недостижимой вершины».
Два функциональных графа считаются изоморфными если они совпадают по структур и по набору аргументов, в точности:
Определение 3: функциональные графы G и G` изоморфны если существует взаимно-однозначная функция g для вершин графов G и G`, такая что для любой вершины v если g(v)=v1, тогда v и v1 терминальные вершины с одинаковыми значениями или нетерминальные вершины с одинаковыми значениями индексов.
Обратите внимание на то, что функциональный граф G имеет только один корень и подграфы любой нетерминальной вершины дифференцированы, изоморфное отображение sigma между G и G` ограничено: корень G должен быть отображен в корень G`, а также все подграфы должны быть отображены в аналогичные. Проверка двух графов на изоморфность довольно простая.
Определение 4: для любой вершины v функционального графа G, подграф с корнем в вершине v определен как граф состоящий из v и всех подграфов.
Лемма 1: если граф G изоморфен графу G` отображением sigma, тогда для любой вершины v в графе G выполнимо: подграф с корнем v изоморфен подграфу с корнем sigma(v).
Доказательство этой леммы очевидно, т.к. ограничения sigma на v и их потомство формирует изоморфное отображение.
Размер функционального графа может быть уменьшен без изменения функции выделением лишних вершин и дублированных подграфов. Граф полученный в результате таких действий и будет нашим исходным графом для представления бинарной функции.
Определение 5: функциональный граф G сокращен если он не содержит вершин v таких, что low(v)=high(v), и не содержит вершин v и v1 таких что их подграфы изоморфны.
Следующая лемма вытекает прямо из определения сокращенного функционального графа.
Лемма 2: Для каждой вершины v в сокращенном функциональном графе, подграф с корнем v будет сам по себе сокращенным.
Следующая теорема доказывает ключевое свойство сокращенного функционального графа, а именно то что такие графы формируют каноническое представление бинарной функции, т.е. каждая функция представлена уникальным сокращенным функциональным графом. В отличие от других канонических представлений бинарных функций, таких как каноническая форма sum-of-product.
Теорема 1: Для любой бинарной функции f , есть один и только один сокращенный функциональный граф выражающий f и любой другой функциональный граф выражающий f будет содержать больше вершин.
Доказательство: Доказательство этой теоремы основано на индукции для размера графа I_f.
Для $/I_f/=0$, f – это константа, 0 или 1. Пусть граф G это сокращенный граф выражающий функцию константу 0. У этого графа не может быть вершин терминальных вершин со значением 1, иначе функция на некотором наборе аргументов даст результат = 1. Предположим, что граф G содержит по крайней мере одну нетерминальную вершину. Тогда, т.к. граф ацикличен должна быть вершина v для которой и low(v) и high(v) будут терминальными вершинами, и значит value(low(v))=value(high(high(v)) = 0. Обе эти вершины достижимы, в этом случае они образуют изоморфный граф или же они идентичны, т.е. value(low(v))=value(high(high(v)). В обоих случаях G не будет сокращенным функциональным графом. Отсюда следует, что только сокращенный граф выражает функцию 0 и он состоит из одной терминальной вершины со значением 0. Аналогично для функции 1.
Следующие предположение, то что теорема для любой функции g с $|I_g| < k$ и, что $|I_f| = k$, где $k>0$. Пусть i минимальное значение I_f , т.е. минимальный набор аргументов от которых зависит функция. Определим функции f_0 и f_1 как $f|_{xi=0}$ и

$fl_{xi=1}$. Обе эти функции имеют зависимый размер меньший k и поэтому представлены уникальными сокращенными функциональными графами. Пусть эти графы G и G`. Мы докажем, что эти графу изоморфны с корневой вершиной со значением i и оба подграфа выражают функции $f_0$ и $f_1$. Пусть v и v` нетерминальные вершины в двух графах такие что index(v)=index(v`)=i. Подграфы с корнями v и v` оба выражают f, т.к. f не зависит от значения x_1, …, x_{i-1}. Подграфы с корнями в вершинах low(v) и low(v`) оба выражают функцию $f_0$ и по правилу индукции должны быть изоморфны некоторому отображению $sigma_0$. Также и подграфы high(v) и high(v`) оба выражают f_1 и изоморфны некоторому отображению $sigma_1$.
Мы сделали утверждение, что подграфы с корнями v и v` изоморфны отображению sigma, определенного как:

	v`	u=v
sigma(u) =	$sigma_0$(u)	u подграф low(v)
	$sigma_1$(u)	u подграф high(v)

Для доказательства этого, нам необходимо показать, что функция sigma достаточно определена и что это изоморфное отображение. Заметим, что если вершина u содержится в обоих подграфах low(v) и high(v), тогда их корни $sigma_0$(u) и $sigma_1$(u) должны быть изоморфны и должны быть изоморфны графу с корнем u и друг другу. Т.к. граф G` не содержит в себе изоморфных подграфов, это возможно только в случае когда $sigma_0$(u)=$sigma_1$(u), и следовательно определение sigma полное и верное. Таким образом то, что sigma это изоморфное отображение, следует прямо из определения. По той же причине, можно заметить, что отображение sigma должно быть однозначное – если существуют достижимые вершины u_1 и u_2 в графе G с sigma(u_1)=sigma(u_2), тогда подграфы с этими вершинами будут изоморфны подграфу sigma(u_1), несмотря на то, что граф G несокращен. Наконец то, что sigma является изоморфным отображением следует прямо из определения и из того факта, что и $sigma_1$ и $sigma_2$ удовлетворяют свойствам.
Аналогично можно сказать, что граф G содержит в себе только одну вершину с index(u)=i т.к. если еще одна такая вершина существует, в графе появятся изоморфные подграфы. Мы сказали, что корнем должна быть вершина v. Предположим, что есть некоторая вершина u с index(u)=j<i , но нет такой вершины w с j<index(w)<i . Функция f не зависит от x_j и поэтому подграфы вершин low(u) и high(u) выражают функцию f, и они равны, т.е. G не сокращенный граф. Аналогично, вершина v` должна быть корнем G` и эти графы изоморфны.
Наконец мы доказали что только сокращенный граф содержит в себе минимальное количество вершин. Предположим, что граф G несокращенный. Тогда мы можем построить граф с меньшим количеством вершин, выражающий ту же самую функцию. Если граф G содержит вершину v такую, что low(v)=high(v), то можно исключить эту вершину, а для всех вершин, которые имеют вершину v в качестве «листа» присвоить значению листа low(v). Если G имеет достижимые вершины v и v` такие что их подграфы изоморфны, то одну из них можно исключить.

III. Свойства

В этой главе мы исследуем эффективность нашего представления на некоторых примерах. На рис. 1 есть несколько сокращенных функциональных графов. Нетерминальные вершины представлены кружками с индексами и двумя путями 0 (low) и 1 (high). Терминальные вершины представлены квадратом со значением.

А. Примеры функций.
Функция, которая в качестве результата дает значение своего аргумента, представлена простой нетерминальной вершиной с двумя путями low со значением 0 и high со значением 1.
Функция «нечет» для n аргументов представлена графом с 2*n+1 вершинами. Легко видеть преимущество такого представления по сравнению с «суммой произведений», для которой требуется 2^n терм.
Следующий пример – это граф выражающий функцию $x_1*x_2+x_4$ и содержащий 5 вершин. Этот пример иллюстрирует ряд ключевых свойств сокращенных графов. Во-первых граф не содержит вершины с индексом 3, т.к. функция не зависит от этого аргумента. Более точно это звучит так: сокращенных граф функции f состоит только из вершин содержащихся в I_f. Также видно что некоторые подграфы пересекаются друг с друг с другом в некоторых вершинах, это значительно сокращает размер представления функции, а также имеет место в реализации алгоритма, если одна операция была представлена в подграфе, результат может быть использован во всех местах пересечения.

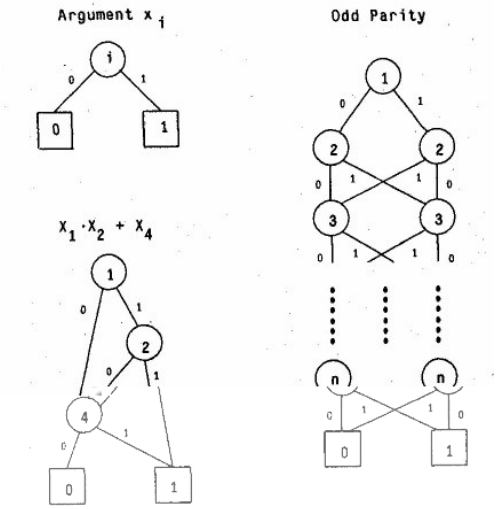


Рисунок 1. Пример функциональных графов

B. Зависимость порядка

Рис. 2 показывает как размер графа зависит от порядка аргументов в функции, даже для эквивалентных функций. Так функции x1*x2+x3*x4+x5*x6 и x1*x4+x2*x5+x3*x6 отличаются только порядком аргументов, но и для представления одной из них требуется граф с 16-ю вершинами, а для другой с 8-ю.

При детальном изучение этих двух графов, можно понять откуда появляется такая проблема. Представьте себе процессор, который вычисляет бинарные функции, производя операции с аргументами в их последовательности, т.е. x1, x2 и так далее, и в конце выдает результат 0 или 1. Такой процессор потребует достаточного количества памяти для хранения информации о аргументах которые он уже получал для правильного вывода результата функции. Некоторые функции требуют хранения малого объема информации. Например, для вычисления функции «равенства», такой процессор требует памяти для хранения значений аргументов, которые он уже получил. Так для функции $x1*x2+\ldots+x_n*x_{n+1}$ процессору понадобиться запомнить была ли хоть одно пара равна 1 и возможно значение предыдущего аргумента, а для функции $x_1*x_{n+1}+\ldots x_n*x_{2n}$, процессору понадобится хранить первые n аргументов для правильного вывода результата. Функциональный граф может рассматриваться как такой процессор. Лучше всего хранить промежуточные результаты в памяти как биты. В таком случае процессору потребуется b битов для кодирования информации о первых i аргументах, в то время как любой граф для этой функции будет содержать по меньшей мере 2^n вершин, как терминальных, так и нетерминальных с индексом больше i, и имеющих входящие ветви их вершин с индексами меньше и равными i. Например, функция $x_1*x_4+x_2*x_5+x_3*x_6$ требует 2^3 ветвей между вершинами с индексами меньше или равными 3 и остальными вершинами. Первые три уровня этого графа формируют полное бинарное дерево, и количество вершин растет экспоненциально по отношению к количеству аргументов.

Посмотрим на это с другой стороны. Определим семейство функций:

$f_{b1,\ldots,bn}(x_{n+1},\ldots,x_{2n})=b_1*x_{n+1}+\ldots+b_n*x_{2n}$

Для всех возможных комбинаций значений $b_1,\ldots,b_n$, а их 2^n, должен быть построен подграф в графе функции $x_1*x_{n+1}+\ldots x_n*x_{2n}$.

При использовании нашего алгоритма необходимо осознавать насколько порядок входных аргументов может повлиять на сложность задачи. Деле мы опишем как решить проблему выбором подходящего порядка.

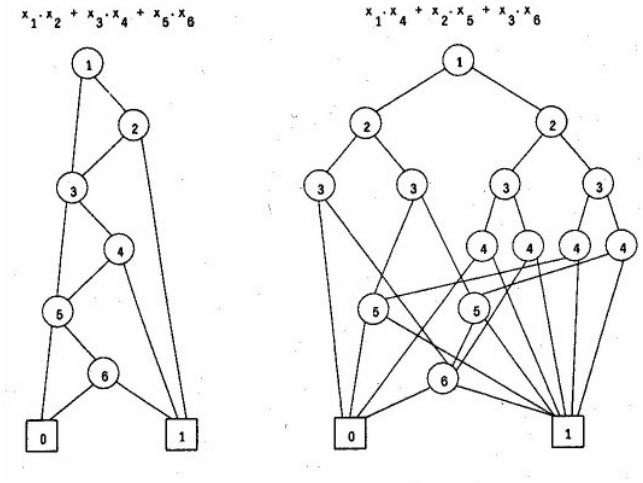


Рисунок 2. Пример функций с зависимостью порядка аргументов

С Наследственная комплексная функция

Некоторые функции не могут быть представлены эффективно при помощи нашего метода в следствии входной последовательности. К сожалению, функции выражающие целочисленное умножение относятся к этому классу. Приложение содержит доказательство того, что для любой входной последовательности $a_1,\ldots, a_n$ и $b_1,\ldots,b_n$ по крайней мере одна из 2n функций представляет целочисленное умножение a*b требует граф содержащий по крайней мере $2^{n/8}$ вершин. Пока эта нижняя граница не столь велика для размеров слов используемых на практике (т.е. 256 для n=64), тем не менее она показывает экспоненциальную сложность этой функции. К тому же, реальная граница еще выше. Опытным путем мы выяснили, что для слов размером n меньше и равном 8, выходная функция умножения требует не более 5000 вершин, для всех вариантов входной последовательности. Тем не менее, для n>10, некоторые результаты требуют граф с более чем 100 000 вершин, и поэтому становится неприменимой на практике.

IV Операции

Рассмотрение программ символической манипуляции как программ выполняющих последовательность команд которые строят представление функции и определение некоторых свойств. Например, мы хотим построить представление функции вычисляемой логический элемент компьютерной сети. Начиная с представления входных данным с помощью графа, мы пройдем через сеть, строя функцию вычисляющую выход на каждом логическом узле применяя узловой оператор к функциям на входе в узел.
Таким образом мы можем проверить ряд свойств функции, равна ли она 1 (тавтология) или 0 (удовлетворение), или другой функции (эквивалентность). Мы также можем получить информацию о множестве удовлетворения.
В этом разделе описаны алгоритмы основных операций с булевой функцией, представленной в виде графа.
В таблице 1 приведены основные операции и их сложность. Функция f представлена графом G, содержит |G| вершин.

Процедура	Результат	Временная сложность
Reduce	G приведен к канонической форме	$O(\|G\|\log(\|G\|))$
Apply	$f_1<op>f_2$	$O(\|G_1\|*\|G_2\|)$
Restrict	$f\|_{xi=b}$	$O(\|G\|\log(\|G\|))$
Compose	$f_1\|_{xi=f2}$	$O(\|G_1\|^2*\|G_2\|)$
Satisfy-one	один элемент S_f	$O(n)$
Satisfy-all	S_f	$O(n*\|S_f\|)$
Satisfy-count	$\|S_f\|$	$O(\|G\|)$

Таблица 1. Основные операции и их сложность

А Структура данных
Представим наши алгоритмы на некотором алгоритмическом языке. Каждая вершина будет представлена как запись вида:

```
type vertex = record
        low, high : vertex;
        index : 1…n+1
        val : (0,1,X)
        id : integer;
        mark : boolean;
end;
```

И нетерминальная и терминальная вершины представлены записью одного типа, но значение полей зависит от типа вершины:

поле	терминальная вершина	нетерминальная вершина
low	null	low(v)
high	null	high(v)
index	n+1	index(v)
val	value(v)	X

Поля id и mark информацию используемую исключительно алгоритмом. Поле id содержит целочисленный идентификатор, уникальный для каждой вершины графа. Порядок этих идентификаторов не важен. Поле mark используется для отметки тех вершин которые были посещены во время обхода графа. Процедура обхода графа представлена на рис. 3, этот алгоритм используется во многих методах, а также, если необходимо произвести некоторое действие с каждой вершиной. Эта процедура начинает обход графа с вершины, и продолжается пока все поля mark не станут равны true или false. Процедура является рекурсивной и обходит все вершины графа, побывав в каждой не более одного раза. Сложность такого алгоритма пропорциональна количеству вершин в графе, т.е. O(|G|). Условием завершения является то, что поле mark у всех вершин становится одинаковым.

```
procedure Traverse(v:vertex);
begin
    v.mark := not v.mark
    …do something to v…
    if v.index ≤ n
    then begin {v nonterminal}
        if v.mark ≠ v.low.mark then Traverse(v.low);
        if v.mark ≠ v.high.mark then Traverse(v.high);
    end;
end;
```

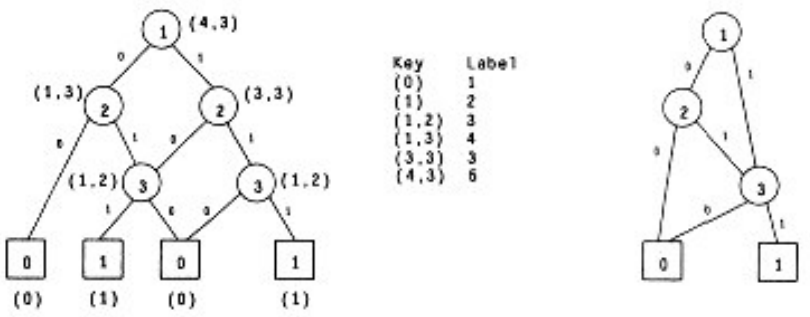
Рисунок 3. Процедура обхода графа

B. Редукция

Алгоритм редукции преобразует произвольный функциональный граф в сокращенный граф, выражающий туже самую функцию. Обработка графа начинается с терминальной вершины к корневой вершине, каждому уникальному подграфу присваивается уникальный идентификатор. Таким образом, что id(v)=id(u) возможно только если $f_v=f_u$. Таким образом получается граф в котором нет повторяющихся поддеревьев.
Совершая обход вершин с терминальной до корневой, алгоритм присваивает метки вершинам следующим индуктивным методом. Во-первых, две терминальные вершины с одинаковым значением должны иметь одинаковую метку. Как пометить все терминальные и нетерминальные вершины индексом больше, чем i, который уже был использован. Так как мы работаем с вершинами, чей индекс равен i , вершина v должна иметь id(v) равный идентификатору, который уже был использован, только при выполнении двух условий. Первое, если id(low(v))=id(high(u)), значит вершина сокращена и можем присвоить id(v)=id(low(v)). Второе, если есть вершина u с индексом index(u)=i и id(low(v))=id(low(u)), и id(high(v))=id(high(u)), тогда графы с вершинами u и v изоморфны и можно установить id(v)=id(u).

```
function Reduce(v: vertex): vertex;
    var subgraph: array[1··|G|] of vertex;
    var vlist: array[1··n + 1] of list;
begin
    Put each vertex u on list vlist[u.index]
    nextid := 0;
    for i := n + 1 downto 1 do
    begin
        Q := empty set;
        for each u in vlist[i] do
            if u.index = n + 1
                then add ⟨key,u⟩ to Q where key = (u.value){terminal vertex}
                else if u.low.id = u.high.id
                        then u.id := u.low.id {redundant vertex}
                        else add ⟨key,u⟩ to Q where key = (u.low.id, u.high.id);
        Sort elements of Q by keys;
        oldkey := (−1;−1);  {unmatchable key}
        for each ⟨key,u⟩ in Q removed in order do
            if key = oldkey
                then u.id := nextid; {matches existing vertex}
                else begin {unique vertex}
                        nextid := nextid + 1; u.id := nextid; subgraph[nextid] := u;
                        u.low := subgraph[u.low.id]; u.high := subgraph[u.high.id];
                        oldkey := key
                    end;
    end;
    return(subgraph[v.id]);
end;
```

Исходный код этой процедуры показан на рисунке 4. Сначала вершины собираются в список по их индексам. Это можно сделать с помощью процедуры обхода. Затем работа ведется со списком от терминальной вершины до корневой. Для каждой вершины в списке мы создаем ключ (value) для терминальной вершины, и (lowid, highid) для нетерминальной вершины, где lowid=id(low(v)) и highid=id(high(v)). Если у вершины lowid=highid, то можно сразу выполнить id(v)=lowid. Используемые вершины упорядочены по ключу. Необходимо пройти по всем вершинам в списке, и отметить вершины с одинаковыми ключами. А также выберем по одной вершине для каждой метки, и сохраним указатели на эти вершины в массиве. Выбранные вершины и дадут нам сокращенный граф. Временная сложность такого алгоритма идентична временной сложности алгоритма сортировки списка. А полученные метки для каждой вершины, в дальнейшем могу быть использованы как уникальные индексы.
На рисунке 5 показан пример работы этого алгоритма. Отметим, что все вершины с индексом 3 имеют одинаковый ключ (key), а вершина с индексом 2 (которая справа) является лишней.



C. Применение (apply)

Процедура Apply создает представление бинарной функции в виде графа. Она создает граф представляющий две функции f_1 и f_2, и оператор <op> (т.е. бинарную функцию 2 аргументов), и создает сокращенный граф f_1<op>f_2, как $[f_1<op>f_2](x_1,\ldots,x_n)=f_1(x_1,\ldots,x_n)<op>f_2(x_1,\ldots,x_n)$.
С помощью нашего представления у нас есть возможность выполнять любые операции с функцией с помощью простого алгоритма. Это выделяет наш метод из других программ для работы с бинарными функциями, которые требуют разные алгоритмы для выполнения умножения, сложения и других действий.
Алгоритм начинает работать с корня, как с функцией двух аргументов, и так до самого конца графа.
Сначала объясним основную идею алгоритма. Потом опишем 2 метода улучшения эффективности алгоритма.
Управляющая структура алгоритма основана на следующей рекурсии
$f_1<op>f_2=(-)x_i*(f_1|_{xi=0}<op>f_2|_{xi=0})+x_i*(f_1|_{xi=1}<op>f_2|_{xi=1})$
Для применения оператора к функции представленной в виде графа с корнями v_1 и v_2 , мы должны рассмотреть несколько случаев. Во-первых, предположим, что эти обе вершины обе терминальные. Тогда результатом будет ничто иное как value(v_1)<op>value(v_2). Но предположим, что хотя одна из этих вершин нетерминальная. Если index(v_1)=index(v_2)=i , мы создаем вершину u и индексом i , и применяем алгоритм рекурсивно для low(v_1) и low(v_2) для получения low(u), и аналогично для high(v_1), high(v_2) для получения high(u).
Предположим, что возможна такая ситуация, когда index(v_1)=i, а v_2 терминальная вершина и index(v_2)>i. Тогда функция представленная графом с корнем v_2 не зависит от x_i , т.е.
$f_2|_{xi=0} = f_2|_{xi=1} = f_2$
Т.к. мы создаем вершину u с индексом i , но рекурсивно применяем алгоритм к low(v_1) и v_2 для получения подграфа с корнем low(u), и к high(v_1) и v_2 для получения – high(u). Аналогичная ситуация если поменять местами v_1 и v_2. В такой ситуации в результате получается несокращенный граф, и поэтому потом необходимо применить алгоритм сокращения.
При использовании описанного алгоритма мы получаем экспоненциальную временную сложность, т.к. при обработке одной вершины, вызывается рекурсивная процедура дважды. Этот алгоритм можно усовершенствовать следующими способами.
Прежде всего, необходимо совершать проход по подграфу не более одного раза. Для этого мы можем определить таблицу содержащую (v_1, v_2, u), т.е. показывающую, что результатом обработки подграфов v_1 и v_2 стал граф с корнем u. В дальнейшем при работе с двумя вершинами, мы сначала проверим наличие этих вершин в таблице. И если эта пара вершин есть в таблице результат можно вернуть сразу. В противном случае, мы совершаем операцию с подграфами и заносим результат в таблицу. Таким образом временная сложность алгоритма снизится к $O(|G_1|*|G_2|)$.
Такая модернизация алгоритма значительно сокращает время работы. Т.к. в графе очень часто встречаются общие подграфы.
Теперь предположим, что один из подграфов, например v_1, это терминальная вершина, и в этом случае значение value(v_1) – это контрольное значение, т.е. мы имеем value(v_1)<op>a=1 для всех a, или value(v_1)<op>a=0 для всех a. Например, 1 – это контрольное значение для всех аргументов OR, а 0 – это контрольное значение для AND. В таком случае нет необходимости производить какие-либо действия. Можно просто создать терминальную вершину с необходимым значением.

```
function Apply(v1, v2: vertex; ⟨op⟩: operator): vertex
    var T: array[1··|G1|, 1··|G2|] of vertex;
    {Recursive routine to implement Apply}
    function Apply-step(v1, v2: vertex): vertex;
    begin
        u := T[v1.id, v2.id];
        if u ≠ null then return(u); {have already evaluated}
        u := new vertex record; u.mark := false;
            T[v1.id, v2.id] := u; {add vertex to table}
            u.value := v1.value ⟨op⟩ v2.value;
            if u.value ≠ X
            then begin {create terminal vertex}
                u.index := n + 1; u.low := null; u.high := null;
            end
            else begin {create nonterminal and evaluate further down}
                u.index := Min(v1.index, v2.index);
                if v1.index = u.index
                    then begin v1 low1 := v1.low; vhigh1 := v1.high
end                 else begin vlow1 := v1; vhigh1 := v1 end;
                if v2.index = u.index
                    then begin v6w2 := v2.low; vhigh2 := v2.high
end                   else begin vlow2 := v2; vhigh2 := v2 end;
                  u.low := Apply-step(ulow1, vlow2);
                  u.high := Apply-step(vhigh1, vhigh2);
              end;
              return(u);
          end;
      begin {Main routine}
          Initialize all elements of T to null;
          u := Apply-step(v1, v2);
          return(Reduce(u));
      end;
```

Рисунок 6. Процедура применения (apply)

Алгоритм показан на рис. 6. Для простоты представления и оптимизации, таблица представлена двумерным массивом. На практике более эффективным оказывается использование хеш-массива. Для проверки, является ли одна из вершин контрольным значением, вычисляется v_1.value<op>v_2.value используя трехзначную алгебру. Так если b<op>1=b<op>0=a, то b<op>X=a, иначе b<op>X=X. Эта техника часто применяется во многих логических симуляторах.

Анализируя временную сложность этого алгоритма, при использовании его на графах с количествами вершин $|G_1|$ и $|G_2|$ соответственно, можно заметить, что процедура apply-step производит рекурсивный вызов только первый раз на данной паре вершин, поэтому общее число рекурсий не может достичь значения $2*|G_1|*|G_2|$. Тем не менее инициализация таблицы требует времени пропорционально размерам графов, т.е. $O(|G_1|*|G_2|)$. Таким образом общая временная сложность определяется как $O(|G_1|*|G_2|)$.

В худшем случае, может потребоваться времени больше. Т.к. граф, полученный в результате применения оператора f_1<op>f_2 может содержать количество вершин равное $O(|G_1|*|G_2|)$. Например, для любых положительных m и n

$f_1(x_1,\ldots,x_{2n+2m})=x_1*x_{n+m+1}+\ldots+x_n*x_{2n+m}$

$f_2(x_1,\ldots,x_{2n+2m})=x_{n+1}*x_{2n+m+1}+\ldots+ x_{n+m}*x_{2n+2m}$

Эти функции представлены графами с $2^{n+m+1} = 0{,}5\ |G_1|*|G_2|$ вершинами. Но в настоящее время нет более эффективного алгоритма для выполнения этой операции, и худший случай ограничен размером результирующего графа.

Итак общая временная сложность в наихудшем случае будет $O(|G_1|+|G_2|+|G_3|)$, где G_3 – это результирующий граф.

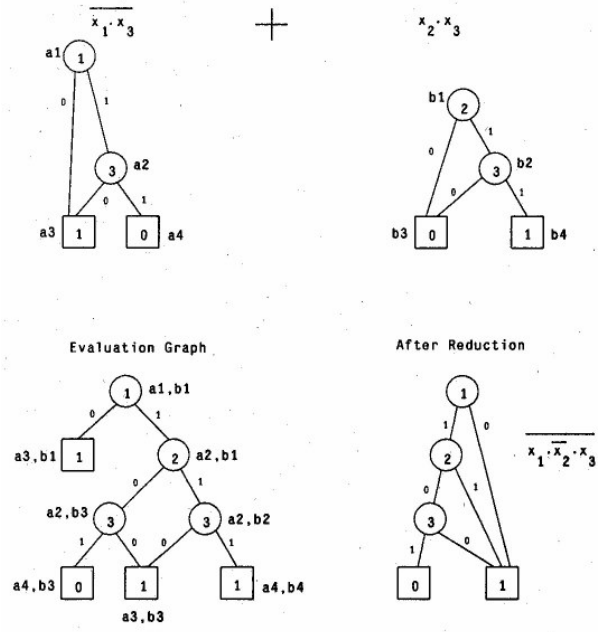


Рисунок 7. Пример работы алгоритма применения

Рисунок 7 показывает, как наш алгоритм выполняет действие OR на двух графах выражающих функции (-)(x_1*x_2) и x_2*x_3. Здесь показан граф до, и после сокращения.

D. Ограничение.
Этот алгоритм преобразовывает граф выражающий функцию f в граф, выражающий $f|_{x_i=b}$. Этот алгоритм проходит по всему графу, начиная с корня и до вершины с индексом i , а потом производится замена high(v)=1 или low(v)=0 для b=1 или b=0 соответственно. Затем вызывается процедура сокращения.
Временная сложность этой операции оценивается временной сложностью процедуры сокращения графа. Этот алгоритм может быть применен одновременно к нескольким константам b, не меняя сложности.

E. Композиция

Алгоритм композиции строит граф, выражающий функцию, полученную как композицию двух других. Композиция может быть представлена с помощью ограничений и бинарных операций, как в следующем разложении (ряд Шаннона)

$f_1|_{xi=f2}=f_2*f_1|_{xi=1}+(-)f_2*f_1|_{xi=0}$

Наши алгоритмы для ограничения и применения достаточны для выполнения композиции. Но тем не менее, емли две функции представлены графами G_1 и G_2 соответственно. То мы получим сложность алгоритма в худшем случае $O(|G_1|^2*|G_2|^2)$. Но мы можем уменьшить ее до $O(|G_1|^2*|G_2|)$ если отметим что $f_1|_{xi=f2}=f_2*f_1|_{xi=1}+(-)f_2*f_1|_{xi=0}$ может быть представлена в нотации ЕТИ (если-то-иначе) ITE (if-then-else)

ITE(a, b, c) = a*b+(-a)*c

Эта операция может быть применена к трем функциям f_2, $f_1|_{xi=1}$, $f_1|_{xi=0}$ при использовании алгоритма apply. Процедура compose сведет эту операцию к композиции двух функций. В этом исходном коде рекурсия compose-step применяет операцию ITE и вычисляет ограничения f_1. Не совсем ясно, имеет ли этот алгоритм сложность 4 порядка от размера первого аргумента. Нам ни попалось ни одного случая, когда сложность алгоритма была бы больше $O(|G_1|*|G_2|)$

Во многих случаях композиция двух функций может быть выполнена гораздо проще. Пусть две функции представлены графами G_1 и G_2 соответственно. Мы сожжем выполнить композицию этих функций, заменяя каждую вершину G_1 с индексом i копией G_2, заменяя все «листья» до терминальной вершины в G_2 «листьями» low(v) и high(v) в зависимости от значения терминальной вершины. Но мы можем сделать это только в том случае когда индекс в полученном графе не нарушит нашу последовательность. Т.е. не может быть индексов $j \in I_{f1}$, $k \in I_{f2}$, таких что i<j<=k или i>j>=k. А в том случае если оба исходных графа упрощены, то и полученный граф будет сокращенным. В силу того, что эта техника применима для ряда случаев, она является стоящей оптимизацией.

```
function Compose(v1, v2: vertex; i:integer): vertex
    var T: array[1..|G1|, 1..|G1|, 1..|G2|] of vertex;
    {Recursive routine to implement Compose}
    function Compose-step(vlow1, vhigh1, v2: vertex): vertex;
    begin
        {Perform restrictions}
        if vlow1.index = i then vlow1 := vlow1.low;
        if vhigh1.index = i then vhigh1 := vhigh1.high;
        {Apply operation ITE}
        u := T[vlow1.id, vhigh1.id, v2.id];
        if u ≠ null then return(u); {have already evaluated}
        u := new vertex record; u.mark := false;
        T[vlow1.id, vhigh1.id, v2.id] := u; {add vertex to table}
        u.value := (¬v2.value·vlow1.value) + (v2.value·vhigh1.value);
        if u.value ≠ X
        then begin {create terminal vertex}
            u.index := n + 1; u.low := null; u.high := null;
        end
        else begin {create nonterminal and evaluate further down}
            u.index := Min(vlow1.index, vhigh1.index, v2.index);
            if vlow1.index = u.index
                then begin vll1 := vlow1.low; vlh1 := vlow1.high end
                else begin vll1 := vlow1; vlh1 := vlow1 end;
            if vhigh1.index = u.index
                then begin vhl1 := vhigh1.low; vhh1 := vhigh1.high end
                else being vhl1 := vbigh1; vhh1 := vbigh1 end;
             if v2.index = u.index
                then begin vlow2 := v2.low; vhigh2 := v2.high end
                else begin vlow2 := v2; vhigh2 := v2 end;
             u.low := Compose-step(vll1, vhl1, vlow2);
             u.high := Compose-step(vlh1, vhh1, vhigh2);
        end;
        return(u);
    end;
begin {Main routine}
    Initialize all elements of T to null;
    u := Compose-step(v1, v1, v2);
    return(Reduce(u));
end;
```

Рисунок 8. Алгоритм композиции

F. Удовлетворение (satisfy)

Довольно часто задаваемый вопрос – это вопрос о том, удовлетворяет ли некоторое множество S_f функции f, включая количество элементов, список элементов или же просто один элемент. Как видно из таблицы 1, сложность этого алгоритма широко варьируется. Простой элемент может быть найден за время пропорциональное n, количеству функциональных аргументов, при условии, что граф сокращен. При условии, что значение элементов S_f это битовые последовательности длины n, алгоритм является оптимальным. Мы можем получить список всех элементов множества S_f за время пропорциональное n, числу элементов. Тем не менее неразумно обрабатывать функции с маленькими графами представления, но с большими множествами удовлетворения S_f. Например функция «константа 1» представлена графом с одной вершиной, а множество удовлетворения содержит все 2^n возможных комбинаций входных данных.

Если мы хотим найти элемент из множества удовлетворения удовлетворяющий свойству, не стоит для этой цели перечислять все элементы множества удовлетворения, и уже потом выбирать из них один элемент с требуемой характеристикой. Вместо этого нам следует записать это свойство в форме бинарной функции, вычислить результат произведение этого свойства и исходной функции, и затем с помощью процедуры satisfy-one выбрать искомый элемент. Мы также можем посчитать размер множества удовлетворения, используя алгоритм, за время пропорциональное размеру графа. На практике оказывается, что гораздо быстрее результат получается при применении этого алгоритма, чем при перечислении всех элементов множества, а уже потом посчитать их количество.

```
function Satisfy-one(v: vertex; var x: array[1··n] of integer): boolean
begin
    if v.value = 0 then return(false); {failure}
```

Рисунок 9. Функция satisfy-one

Процедура satisfy-one показана на рисунке 9, в качестве аргументов получает сам граф (указатель на вершину) и массив x. В качестве результата она возвращает false если множество удовлетворения пусто, или true в противном случае и массив x содержит значение элемента множества удовлетворения. Схема работы процедуры основана на первоначальном «спуске» к терминальной вершине со значением 1, и затем построения обратного пути к корню. Эта процедура работает с любым функциональным графом. При применении этой процедуры к несокращенному графу, может потребоваться довольно много времени. А для сокращенного графа мы можем определить следующее свойство.

Лемма 3: каждая нетерминальная вершина сокращенного функционального графа имеет наследственную терминальную вершину со значением 1.

```
procedure Satisfy-all(i: integer; v: vertex; x: array[1··n] of integer):
begin
    if v.value = 0 then return; {failure}
    if i = n + 1 and v.value = 1
    then begin {success}
        Print element x[1], ···, x[n];
        return;
    end;
    if v.index > i
    then begin {function independent of x_i}
        x[i] := 0; Satisfy-all(i + 1, v, x);
        x[i] := 1; Satisfy-all(i + 1, v, x);
    end
    else begin {function depends on x_i}
        x[i] := 0; Satisfy-all(i + 1, v.low, x);
        x[i] := 1; Satisfy-all(i + 1, v.high, x);
    end;
end;
```

Рисунок 10. Функция satisfy-all

Для перечисления элементов множества удовлетворения, мы модем сделать полную проверку графа, и выводить элемент множества, как только достигли вершины со значением 1. Процедура satisfy-all, показанная на рисунке 10, реализовывает этот метод. Эта процедура содержит три аргумента: индекс текущего функционального элемента в

перечислении, корневую вершину обрабатываемого подграфа и массив, описывающий состояние поиска. Процедура вызывается с корневой вершины графа, а массив содержит случайные

V Экспериментальные результаты

Как и все другие алгоритмы для решения NP-сложных задач, наш алгоритм имеет худший случай представления неприемлемым для всех проблем, кроме наименьших. Мы надеемся, что наш подход окажется применим не практике. Мы уже показали, что размер графа сильно зависит от порядка входных параметров, и то что наш алгоритм эффективен, если функция представлена графом разумного размера. Главный вопрос нашего эксперимента: как выбрать подходящий порядок входных параметров, и насколько большим может быть граф на практике.

Мы сказали, что наши алгоритмы применимы для решения задач логической верификации, комбинаторики. В целом наш опыт показал, что анализируя проблему можно правильно и эффективно выбрать порядок входных параметров. Но стоит ли тратить на это время. На практике, алгоритм довольно быстро справляется с графами содержащими 20000 вершин.

В этой статье мы обсудим проблему проверки удовлетворения выполнения логической функции ее спецификации. В качестве примера мы рассмотрим семейство АЛУ 74181 и 74182 TTL. 181 представляет собой 4-битное АЛУ, а 182 - ???. Комбинируя эти чипы можно получить АЛУ работающего со словом любой длины, кратной 4-ем. АЛУ с размером слова n имеет 6+2n входов: 5 контрольных входов помеченных как m, s_0, s_1, s_2, s_3 для выбора функции АЛУ, бит переноса cin, и 2 информационных слова по n бит каждое, обозначенные $a_0,\ldots,a_{n-1}$ и $b_0,\ldots,b_{n-1}$. На выходе n+2 бит: n бит результата $f_0,\ldots,f_{n-1}$, бит переноса – cout, и бит сличения A=B (результат применения AND к результату).

Для нашего эксперимента мы вывели функцию для двух чипов и объединили их для получения различных комбинаций АЛУ. Таким образом мы получили АЛУ с размером слова 4, 8, 16, 32 и 64 бита (таблица 2). Эти данные основаны на наилучшей входной последовательности, которую удалось найти. В таблице pattern показывает количество различных входных комбинаций. Процессорное время замерялось с использованием Digital Equipment Corporation VAX 11/780 (1-MIP машина). Последняя колонка показывает размер графа для выполнения A=B. Из всех случаев, здесь граф оказался самым большим.

Как видно, время требуемое для такой реализации довольно разумно, т.к. основные процедуры работают довольно быстро. Время требуемое для сокращения графа, отведенное на обработку интерфейса, на вызовы функций apply-step и compose-step составляет порядка 3 ms. Например для вычисления 64-битного АЛУ последние две процедуры вызывались более $1{,}6*10^6$ раз. Общая сложность представления составляет квадрат размера слова.

И размер графа и число логических элементов растет линейно в зависимости от размера слова, а время растет как произведение этих двух факторов. При всестороннем анализе зависимость была бы экспоненциальной.

На рисунке 11 видно как размер графа зависит от входной последовательности. Наилучший вариант №1. Подобную ситуацию мы уже рассматривали в 3-ей главе. Немного менее эффективным порядком оказался способ №2, как видно мы поменяли порядок следования битов (от старшего к младшему). Наихудшим вариантом оказался способ №4, здесь виден экспоненциальный рост.

Эксперимент показал, что наш подход является эффективным для обработки логических функций, с одним единственным ограничением. Необходимо правильно выбрать порядок входных параметров. Правильный выбор заметно ускорит обработку. По сравнению же с другими представлениями бинарных функции наш выбор особенно эффективен.

Например, таблица истинности неприменима для представления АЛУ с длиной слова больше 8.

VI Выводы

Мы показали, что если взять уже известное графическое представление бинарных функций, наложив ограничения на вершины, то граф минимального размера представляющий функцию будет канонической формой. Более того, мы можем привести любой граф, выражающий какую-либо функцию, к канонической форме за линейное время. Наш алгоритм сокращения не только минимизирует граф за довольно малое время, но также с легкостью выполняет проверку на эквивалентность, удовлетворенность или тавтологию. Это свойство оказалось очень полезным на практике.

Мы представили множество алгоритмов для выполнения различных операций с бинарными функциями, представленными в виде графов. Комбинирую техники работы с бинарными функциями и работы с графами, мы достигли высоких результатов. В нашем случае в качестве ограничения выступает само представление функции (т.е. количество памяти необходимое для хранения вершин), а не операции над ним.

Акерс изобрел способ для сокращения двоичной диаграммы решений выходящей функции для системы. Например, он представил все функции для всех 8 выходов АЛУ 74181 35-ю вершинами, а наше представление потребовало 918. Ряд таких техник может быть применимо к нашему представлению.

Большинство цифровых систем имеют многоканальные выходы. В нашем случае мы представили каждую выходящую функцию отдельным графом, несмотря на то что функции могут быть очень похожи, или даже содержать изоморфные подграфы. В качестве альтернативы, можно представить множество простых графов с кратным корнем. Наш алгоритм сокращения может быть применен к таким графам для исключения дублированных подграфов. Например, мы можем представить n+1 функции для добавления двух n-битных чисел простым графом с 9n-1 вершинами, в то время как отдельный граф содержал бы в себе $3n^2+6n+2$ вершин. Исходя их этой идеи мы можем определить множество графов, как простую структуру данных, и использовать алгоритм сокращения для получения нового графа. С такой структурой, мы можем улучшить процедуру apply представив в качестве аргументов (v_1, v_2, <op>, u), таким образом чтобы результат операции заносился в граф с корнем u.

Акерс также экономит ресурсы (память) представляя функцию как декомпозицию двух. Т.е. функция $f|_{xi=g}$ может быть представлена как f и g (которые в свою очередь тоже могут быть представлены в декомпозиционной форме). К сожалению множество функций имеют довольно много вариантов декомпозиции и это не приводит к канонической форме. Но тем не менее, как видно из раздела IV-E есть ряд функций f и g, которые при композиции просто заменяют вершины графа, выражающего одну функцию, графами, выражающими другую. Для такого случая функции могут храниться в декомпозиционной форме и приводится к канонической форме динамически во время обработки.

Например функция $x_1*x_{2n}+x_2*x_{2n-1}+\ldots+x_n*x_{n+1}$ потребует граф с 2^{n+1} вершинами. Эта функция может быть представлена как ряд функций $f_n=x_n*x_{n+1}$:

$f_i=(x_i*x_{2n-i+1}+x_{i+1})|_{x(i+1)=f(i+1)}$

для n>i>=1. Каждая такая функция может быть представлена 6-ю вершинами.

К сожалению еще не ясно, как часто встречаются такие случаи и как их выявить.

Приложение

Сложность целочисленного умножения

В этом приложении мы докажем, что наше представление не применимо для функций выражающих целочисленное умножение, т.к. размер графа растет экспоненциально. Предположим, что всего (2n)! возможных комбинаций входных параметров, в данной ситуации мы можем не надеяться на получение экспериментального результата, и мы должны доказать это.

Нам необходимо доказать не только, то что для выполнения умножения большие объемы информации должны быть перенесены из множества входов в множество выходов, но и то, что отдельные результат требует большой передачи данных.

Предположим, производится умножение $a_1,\ldots,a_n$ и $b_1,\ldots,b_n$, соответствующих бинарному представлению целых чисел a и b с a_1 и b_1 в качестве младших разрядов. Результат будет содержать 2n выходов, исходя их определения произведения a*b, как функции $mul_i(a_1,\ldots,a_n,b_1,\ldots,b_n)$ для 1<=i<=n. Для перестановки π {1, … , 2n}, пусть G(I, π) – это граф с входами $x_1,\ldots,x_{2n}$ определяющий функцию $mul_i(x_{\pi(1)}, \ldots , x_{\pi(2n)})$.

Теорема 2: Для любого pi существует i, такое, что 1<=i<=2n и G(i, π) содержит по меньшей мере $2^{n/8}$ вершин.

Доказательство: Если один их сомножителей кратен 2, то результатом будут биты второго сомножителя с некоторым смещением, определяем степенью числа 2. Например, если $b=2^j$, то

$$[a*b]_i = \begin{cases} a_{i-j} & j<i<=j+n \\ 0 & \text{иначе} \end{cases}$$

Граф G(i, π) должен содержать достаточное количество вершин для представления всех значений этого произведения. Более того мы можем показать, что для любой входной последовательности, мы можем выбрать какой из входящих сомножителей контролирующий, а какой информационный. Это встречается по меньшей мере n/8 раз.

Более формально это выглядит так, пусть для перестановки π

$t=|\{j|1<=j<=n, \pi(j)<=n\}|$,

т.е. количество битов множителя a в первой половине входной последовательности. Если t>=n/2 определим множества F и L как

$F=\{\pi(j)|1<=j<=n, \pi(j)<=n\}$

$L=\{\pi(j)|n+1<=j<=2n, \pi(j)>n\}$

Множество F представляет те индексы множителя a, которые появятся в первой половине входной последовательности, а множество L определяет мндесы b, которые появятся во второй половине.

Если t<n/2 тогда множества F и L будут

$F=\{\pi(j)|1<=j<=n, \pi(j)>n\}$

$L=\{\pi(j)|n+1<=j<=2n, \pi(j)<=n\}$

т.е. наоборот.

И в первом и во втором случаях множества содержат по меньшей мере n/2 элементов. Пусть элементы множества F будут информационными битами, а элементы множества L контролирующими (т.е. битами смещения).

Для 1<=i<=2n-1 определим множество F_i как

$F_i=\{j|\exists\ j \in F, k \in L(j+k=i+n+1)\}$

и пусть $q_i=|F_i|$. Это выход i, F_i представляет те индексы информационных битов в первой половине входящей последовательности в соответствии с контролирующими битами во второй половине.

Теперь определим множество

$S_i=\{x_1, \ldots, x_n|x_j=0$ если $\pi(j)$ не принадлежит $F_i\}$

Это множество содержит 2^{qi} возможных значений для первых n входов. Мы определили, что G(I, π) должен содержать уникальные вершины для каждого элемента S_i. В противном случае можно было бы выбрать две различные последовательности входов $x_1,\ldots,x_n$ и $x_1`,\ldots,x_n`$, ведущих к одной и той же вершине графа G(I, π) для значения j, $\pi(j) \in F_i$ и $x_j <> x_j`$. Теперь определим последовательности $x_{n+1},\ldots,x_{2n}$ и $x_{n+1}`,\ldots,x_{2n}`$ как

$$x_k=x_k` = \begin{cases} 1, & \pi(j)+\pi(k)=i+n+1 \\ 0, & \text{иначе} \end{cases}$$

Отметим, что x_k и $x_k`$ равны 1 только для одного значения k. Эти последовательности ведут к одной терминальной вершине в графе G(I, π), но

$mul_i(x_{\pi(1)}, \ldots , x_{\pi(2n)})=x_j$

пока

$mul_i(x_{\pi(1)}`, \ldots , x_{\pi(2n)}`)=x_j` <> x_j$

Это противоречие приводит нас к выводу, что граф должен содержать по меньшей мере 2^{qi} вершин.

Для получения окончательного результата, нам необходимо показать, что для некоторого значения i, $q_i>=n/8$.

Лемма 4: пусть A, B есть подмножества {1,…,n}, каждое из которых содержит по меньшей мере n/2 элементов. Для 1<=i<=2n-1

$q_i=|\{<a,b>|a \in A, b \in B, a+b=i+1\}|$

тогда существует значение i, такое что $q_i>=n/8$.

Доказательство: заметим, что A, B множества каждое из которых содержит по меньшей мере n/2 элементов. И по меньшей мере $n^2/4$ пар <a,b> таких что $a \in A$, $b \in B$. Тогда

$$\sum_{j=1}^{2n-1} q_i \geq \frac{n^2}{4}$$

Для некоторого значение i, q_i должно быть по меньшей мере не меньше среднего значения q_j

$$q_i \geq \frac{1}{2n-1} \cdot \frac{n^2}{4} \geq \frac{n}{8}$$

Эта теорема показывает, что для любой последовательности входов, некоторые произведения будут экспоненциального размера. Это оставляет возможность того, что для некоторой последовательности может быть построен граф полиномиального размера. Но это предположение не оправдано, любой граф будет экспоненциального размера, несмотря на входящую последовательность.