Библиотека многокорневых бинарных решающих диаграмм BddFunctions и её применение*

Д. Ю. Бугайченко
DmitryBugaychenko@math.spbu.ru
И. П. Соловьев
soloviev@is1483.spb.edu

В данной работе мы предлагаем структуры данных для представления конечнозначных функций и матриц — многокорневые бинарные решающие диаграммы ($MRBDD$), а также алгоритмы выполнения некоторых операций (сумма, произведение, суперпозиция, сравнение) над объектами в этом представлении. За счет более эффективного повторного использования элементов структуры многокорневые бинарные решающие диаграммы обеспечивают более компактное представление по сравнению с широко распространенными многотерминальными бинарными решающими диаграммами ($MTBDD$), что подтверждается экспериментальными результатами. Предложенный подход реализован авторами в рамках библиотеки BddFunctions, предоставляющей гибкий объектно-ориентированный C++-интерфейс для работы с функциями, матрицами и множествами.

*Работа выполнена при поддержке гранта РФФИ 09-01-00525-а.

Введение

В современной информатике существует обширный класс задач, при решении которых необходимо оперировать большими множествами, функциями или матрицами. К таким задачам можно отнести верификацию программного и аппаратного обеспечения [2], автоматическое планирование [10], обработку образов и т. д. Обычные описывающие подходы к представлению данных (например, перечисление элементов) в таких задачах ведут к неприемлемому расходу памяти, тогда как конструирующие подходы (например, определение функции в виде набора правил) сильно усложняют алгоритмы манипуляции с такими объектами, особенно в алгоритмах с поиском неподвижной точки.

Часто применяемой на практике альтернативой описывающим и конструирующим подходам является представление данных с помощью *решающих диаграмм* [5]. В этом случае представляемый объект (множество, функция или матрица) кодируется в виде рекурсивной графовой структуры в соответствии с определенными правилами. При правильно настроенных параметрах кодирования во многих случаях можно добиться того, что размер полученного графа будет расти значительно медленнее, чем размер соответствующего перечисляющего описания. Например, во многих случаях решающая диаграмма булевой функции n переменных ($f : \{0,1\}^n \to \{0,1\}$) имеет размер порядка $O(n)$, тогда как представление этой же функции в виде таблицы истинности потребует порядка $O(2^n)$ памяти.

В то же время операции над объектами, представленными в виде решающих диаграмм, выполняются за полиномиальное (как правило, не более чем за квадратичное) время от размера графов. При этом размер полученного в результате операции графа зависит не от размера исходных графов, а от структуры полученного объекта (полученный в результате вычисления некоторой операции граф может быть как больше, так и меньше исходных графов). Кроме того, при зафиксированных параметрах кодирования решающие диаграммы являются *каноническим* представлением (одному объекту может соответствовать только одна кодировка). Эти особенности позволяют с успехом применять решающие диаграммы там,

где применение других конструирующих подходов (например, описание булевой функции с помощью логических связок $\neg$, $\wedge$ и $\vee$) не применимы. Например, для решения уравнений вида $f(x) = x$, где $f : \{0,1\}^n \to \{0,1\}^n$ (часто возникающая задача при проведении автоматической верификации) использование стандартных конструирующих подходов приведет к полному перебору входных значений за экспоненциальное время, тогда как для решающих диаграмм предложено несколько итеративных алгоритмов, в большинстве случаев сходящихся за полиномиальное время.

По своей сути решающие диаграммы напоминают деревья принятия решений [16], но они обладают важной отличительной чертой — одна и та же вершина может иметь несколько «родительских» вершин, что позволяет эффективно использовать повторно фрагменты диаграммы с идентичной структурой. В результате, решающая диаграмма «подстраивается» под структуру представляемого объекта, проявляет его внутренние закономерности и использует их для оптимизации представления (однако происходит это только при правильно подобранных параметрах кодировки).

Существует несколько разновидностей решающих диаграмм. В задачах, связанных с представлением и анализом булевых функций и конечных множеств, широкое распространение получили бинарные решающие диаграммы (BDD) [5]. В тех же задачах, где приходится иметь дело с конечнозначными функциями или нечеткими множествами, часто применяются *многотерминальные* (алгебраические) бинарные решающие диаграммы ($MTBDD$) [7] и различные их модификации.

Изначально бинарные решающие диаграммы предназначались для эффективного представления булевых функций вида $f : \{0,1\}^n \to \{0,1\}$, но позже были адаптированы и для конечнозначных функций вида $f : \{0,1\}^n \to S$, где S есть некоторое конечное непустое множество. Формально, упорядоченная бинарная решающая диаграмма — ориентированный корневой ациклический граф с множеством вершин $V = N \cup T$, где $N \cap T = \emptyset$. Вершины множества N являются *нетерминалами*, и для каждой такой вершины $v \in N$ определены *значение порядка* $index(v) \in \{1, \ldots, n\}$ и ровно две дочерние вершины $low(v), high(v) \in V$. Индексы нетерминальных вершин i соответствуют аргументам определяемой функ-

ции x_i. Вершины множества T являются *терминалами* и не имеют дочерних вершин. Для таких вершин $v \in T$ определено значение $value(v) \in S$. Кроме того, выполнено *условие порядка*: для любого нетерминала $v \in N$ и любой его дочерней вершины v' выполнено либо $v' \in T$, либо $index(v) < index(v')$. Каждой вершине $v \in V$ сопоставим функцию n переменных $f_v : \{0,1\}^n \to S$ следующим образом:

- если $v \in T$, то $f_v(x_1, \ldots, x_n) \doteq value(v)$;
- если $v \in N$ и $index(v) = i$, то

$$f_v(x_1, \ldots, x_n) \doteq \begin{cases} f_{high(v)}(x_1, \ldots, x_n), \text{iff } x_i = 1 \\ f_{low(v)}(x_1, \ldots, x_n), \text{iff } x_i = 0. \end{cases}$$

Заметим, что функции вида $f : \{0,1\}^n \to \{0,1\}$ задают подмножества $[f]$ на множестве $\{0,1\}^n$ следующим образом:

$$[f] = \{x \in \{0,1\}^n | f(x) = 1\}.$$

При этом стандартные операции с функциями ($\wedge$, $\vee$ и т. д.) эквивалентны соответствующим операциям над множествами ($\cap$, $\cup$ и т. д.).

С точки зрения практического применения наибольший интерес представляют *редуцированные* решающие диаграммы, в которых не существует двух *различных* вершин, представляющих *одинаковые* функции. Алгоритм редукции бинарных решающих диаграмм предложен в [5]. Редуцированные решающие диаграммы содержат меньшее количество вершин за счет повторного использования эквивалентных блоков, кроме того, они являются канонической формой представления при условии зафиксированного порядка переменных.

Простыми диаграммы называются в том случае, если множество S совпадает с $\{0,1\}$, а *многотерминальными* — если S есть конечное множество с более чем двумя элементами, например, подмножество целых чисел вида $\{0, \ldots, 2^n - 1\}$ или конечное подмножество рациональных чисел. Простые диаграммы используются для представления булевых функций ($f : \{0,1\}^n \to \{0,1\}$), а многотерминальные обобщают это представление для конечнозначных функций ($f : \{0,1\}^n \to S$). В редуцированных диаграммах одному элементу множества S соответствует не более одной терминальной вершины $v \in T$, следовательно, в простых диаграммах может

быть только два терминала, тогда как в многотерминальных — до $|S|$ терминалов[1].

Альтернативой расширения множества терминальных вершин для представления конечнозначных функций является введение нескольких корневых вершин. Бинарная решающая диаграмма с k корневыми вершинами $\{v_j\}_{j=1}^{k}$ представляет собой набор из k булевых функций $f_{v_j} : \{0,1\}^n \to \{0,1\}$. Этот набор, в свою очередь, задает функцию $f : \{0,1\}^n \to S$ следующим образом:

$$f(x_1, \ldots, x_n) \doteq bin_S(f_{v_1}(x_1, \ldots, x_n), \ldots, f_{v_k}(x_1, \ldots, x_n)),$$

где $bin_S : \{0,1\}^k \to S$ есть некоторая *двоичная кодировка* множества S. Например, для представления функций вида $f : Z_n \to Z_n$, где $Z_n = \{0, \ldots, 2^n - 1\}$ есть подмножество неотрицательных[2] целых чисел, можно использовать стандартную двоичную кодировку для целых чисел $bin_{Z_n}(x_1, \ldots, x_n) = \sum_{j=1}^{n} x_j * 2^{j-1}$. В этом случае функцию f можно представить в виде упорядоченного набора булевых функций $\{f_{v_j} : \{0,1\}^n \to \{0,1\}\}_{j=1}^{n}$ следующим образом:

$$f(x) = bin_{Z_n}(f_{v_1}(bin_{Z_n}^{-1}(x)), \ldots, f_{v_n}(bin_{Z_n}^{-1}(x))),$$

где $bin_{Z_n}^{-1} : Z_n \to \{0,1\}^n$ есть обратная к bin_{Z_n} функция, строящая битовый вектор по данному значению $x \in Z_n$. Следовательно, функцию f можно представить в виде n-корневой бинарной решающей диаграммы $\{v_j\}_{j=1}^{n}$.

Преимуществом *многокорневых* бинарных решающих диаграмм ($MRBDD$) является их меньший размер, достигаемый в основном за счет более эффективного повторного использования фрагментов одинаковой структуры. Количество возможных нетерминальных вершин $v \in N$ с индексом $index(v) = i$ равно количеству размещений A_2^C, где C есть количество возможных дочерних вершин

[1] Заметим, что решающие диаграммы называют *бинарными* в связи с количеством дочерних вершин у нетерминалов, а не из-за количества терминальных вершин.

[2] Для простоты изложения выбраны неотрицательные целые числа, однако предложенные алгоритмы без модификаций работают и для отрезков вида $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$.

вершины v. С учетом условия порядка C есть сумма количества возможных терминальных вершин и количества возможных нетерминальных вершин $v' \in N$, таких что $index(v') > i$. Обозначим количество терминальных вершин как $|T|$. Количество возможных нетерминалов с максимальным индексом $i = n$ в этом случае равно $A_2^{|T|} = |T| * (|T| - 1) = |T|^2 - |T|$. Далее, по индукции вычисляется и количество возможных нетерминальных вершин с индексом i, равное $|T|^{2^{n-i+1}} - |T|^{2^{n-i}}$. При применении многокорневых решающих диаграмм количество терминалов постоянно и равно 2, тогда как для многотерминальных диаграмм $|T| = |S| \approx 2^k$, что и ведет к существенно большей вариативности структуры и меньшей степени повторного использования.

С другой стороны, преимуществом многотерминальных диаграмм является простота[3] реализации операций над функциями, достигающаяся за счет того, что множество S представлено явно. В случае же многокорневых диаграмм необходимо предварительно определить соответствие некоторой операции на множестве S набору операций над элементами двоичной кодировки. Для большинства представляющих интерес множеств (отрезки целых чисел и конечные подмножества рациональных чисел) подобное соответствие уже определено для операций над отдельными элементами [8], однако его обобщение для работы с функциями реализовано не было.

В данной работе предлагается разработанная авторами библиотека BddFunctions, реализующая алгоритмы выполнения стандартных операций над целочисленными функциями и матрицами, представленными в виде многокорневых бинарных решающих диаграмм (*раздел 1.*), а также приводятся экспериментальные данные, позволяющие сравнить эффективность предложенного представления с аналогичным представлением в виде многотерминальных решающих диаграмм (*раздел 2.*).

[3]Под простотой здесь мы понимаем в первую очередь простоту разработки алгоритма, а не его алгоритмическую сложность.

1. Библиотека BddFunctions

Реализация алгоритмов, основанных на решающих диаграммах сопряжена с решением большого количества технических и инфраструктурных задач, общий объем которых легко может превысить объем работ по реализации основного алгоритма. К счастью, существует несколько доступных библиотек — *пакетов решающих диаграмм* [9] — в которых основная часть технических и инфраструктурных задач уже решена. Наибольшее распространение получили пакеты Colorado University Decision Diagram Package (CUDD) [15] и BuDDy [13]. В данной работе мы остановили свой выбор на пакете CUDD, т. к. он реализует работу и с классическими, и с многотерминальными бинарными решающими диаграммами.

Следует заметить, что даже применение пакета решающих диаграмм не позволяет разработчику полностью уйти от решения инфраструктурных задач. Программный интерфейс, предоставляемый большинством существующих пакетов, реализован на достаточно низком уровне абстракции. В результате даже самые простые действия (завести целочисленную переменную, построить тождественную функцию и т. д.) требуют написания достаточно большого объема кода, что, как правило, вынуждает разработчиков вводить дополнительный промежуточный слой, позволяющий работать на более высоком уровне абстракции.

Разработанная нами библиотека BddFunctions является именно таким промежуточным звеном. Основными понятиями программного интерфейса библиотеки являются *тип данных*, *переменная*, *функция*, *матрица* и *множество* (более подробно архитектура и внешний интерфейс библиотеки будут рассмотрены в *разд. 1.1.*). Для всех этих понятий в рамках библиотеки реализованы все базовые операции и преобразования, предоставлена возможность реализовать дополнительные операции, а все детали технической реализации скрыты от конечного пользователя, что позволяет сразу начинать разработку приложений в терминах более высокого уровня.

Основной код программного интерфейса библиотеки разработан на GNU C++ с использованием компилятора MinGW и интегрированной среды разработки Code::Blocks [6]. Основные же алгоритмы,

описанные в *разд. 1.2.*, реализованы на GNU C с целью обеспечения максимальной производительности.

1.1. Архитектура библиотеки

Основные классы, посредством которых пользователям доступна функциональность библиотеки, представлены на рис. 1. Начало работы с системой осуществляется через класс `Manager`, который хранит всю инфраструктурную информацию и позволяет создавать другие объекты. Следует отметить, что каждый экземпляр класса `Manager` создает автономное рабочее пространство, следовательно, объекты, созданные с помощью одного экземпляра, могут свободно взаимодействовать друг с другом, но не могут взаимодействовать с объектами, созданными другим экземпляром менеджера.

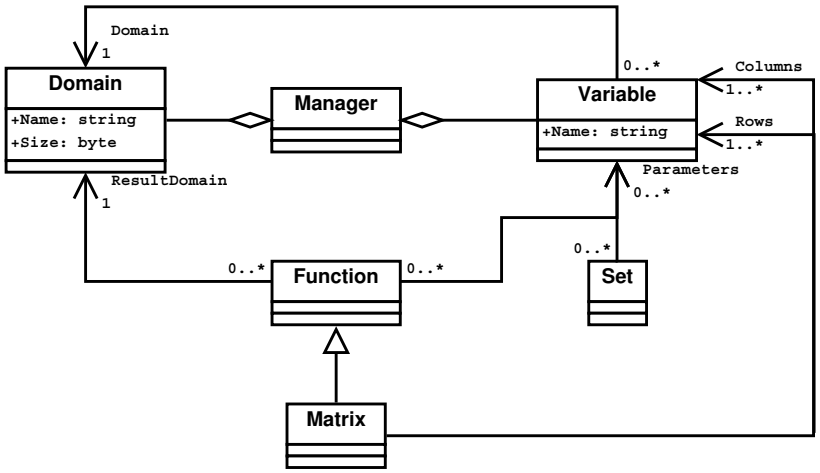


Рис. 1. Основные классы библиотеки

Начинать работу с менеджером следует с создания *доменов* (`Domain`). Домен определяет некоторый тип данных, включая способ представления этого типа в виде набора битов, размер битового представления, а также соответствие стандартных операций

над объектами домена операциям с их двоичным представлением. Каждый домен регистрируется в системе под определенным именем, которое используется далее для ссылок на этот домен.

После регистрации доменов следующим шагом является создание *переменных* (`Variable`). Переменная определяет отдельный параметр данного типа (домена), от которого может зависеть некоторая функция. Переменные, как и домены, при регистрации получают уникальные имена для ссылок.

Когда все необходимые переменные зарегистрированы, перед началом основных вычислений следует инициализировать инфраструктуру пакета решающих диаграмм, вызвав метод `InitializeCudd`, что приведет к созданию всех необходимых BDD-переменных и связыванию их с высокоуровневыми переменными. При инициализации BDD-переменных используется классический «чередующийся» порядок переменных[4], что обеспечивает достаточную компактность представления в большинстве случаев.

После завершения инициализации библиотека готова к работе и можно начинать строить необходимые *функции*, представленные в библиотеке классом `Function`. Каждая функция характеризуется типом возвращаемого значения, набором параметров, а также таблицей значений, закодированной в виде многокорневой бинарной решающей диаграммы. Выполнение операций с функциями осуществляется посредством переопределенных для класса `Function` стандартных арифметических операторов, что обеспечивает максимальную простоту и наглядность кода. Заметим, что выполнение операций с функциями возможно, только если они принадлежат к одному домену (возвращают результаты одинакового типа), однако предусмотрена возможность смены домена посредством определения соответствия битовых представлений.

Функции можно создавать, выполняя операции над другими функциями, кроме того, есть возможность создавать *постоянные функции* (функции, возвращающие одно и то же значение вне зависимости от параметров: $f(x) = c$) и *тождественные функции* (функции, возвращающие значение параметра: $f(x) = x$), кото-

[4]Порядок переменных может оказать существенное влияние на размер решающей диаграммы [5].

рые и используются в качестве основных блоков для описания более сложных функций. Для создания постоянных функций можно воспользоваться методом `GetConstantFunction` или оператором «()» класса `Domain`. Для создания тождественных функций предназначены метод `GetIdentityFunction` класса `Variable` и оператор приведения к `Function` (последний позволяет использовать объекты класса `Variable` в тех же выражениях, что и объекты класса `Function`). Выполнение базовых операций с библиотекой `BddFunctions` может выглядеть так:

```
// Создаем менеджер
Manager manager;

// Регистрируем домен
Domain byte = manager.RegisterDomain(
    "byte", 8, IntegerOperations::GetInstance());

// Регистрируем переменные
Variable x = manager.RegisterVariable("x", byte);
Variable y = manager.RegisterVariable("y", byte);

// Инициализируем инфраструктуру CUDD
manager.InitializeCudd();

// Вычисляем сумму двух тождественных функций,
// умноженную на константу
Function f = (x + y) * byte(3);
```

Расширением понятия функции является понятие *матрицы*, реализованное в классе `Matrix`. Основное отличие матрицы от функции заключается в том, что множество аргументов разбито на два непересекающихся подмножества: ряды и колонки. Это разбиение используется для организации матричного умножения в соответствии с правилами. Создать матрицу можно, используя существующее описание функции, указав, какие переменные следует применять для индексирования колонок, как показано далее:

```
// Вычисляем функцию от переменных x и y, на
// основании которой будет создана матрица
```

```
Function f = x + y;

// Создаем матрицу, указывая использование переменной x
// для индексирования столбцов и y - для строк
Matrix m(f, VariableVector(1, x));
```

Достаточно часто в задачах, связанных с автоматической верификацией, возникает необходимость моделирования кусочно-заданных функций. Для предоставления этой возможности в библиотеке реализовано понятие *множества* (`Set`), как ограничения значений некоторых переменных. Изначально множества возникают при сравнении значений функций, после чего их можно комбинировать, используя операторы &, | и !, а для построения кусочно-заданной функции можно использовать операцию *if-then-else* (`Ite`). Следующий пример демонстрирует работу с множествами для описания функции $\max(x, y)$:

```
Function max = (x > y).Ite(x, y);
```

Одной из важных точек расширения библиотеки является возможность определения собственных типов данный, двоичных кодировок и операций. Классы и интерфейсы, обеспечивающие это расширение, представлены на рис. 2.

Каждый домен связан с двумя специальными объектами: *конвертером* (`Converter`), используемым для преобразования отдельных типизированных объектов (как правило, чисел) в двоичную кодировку и обратно, и *операциями* (`IBddOperations`), описывающими реализацию стандартных операций в двоичной кодировке: сложение, умножение, матричное умножение, сравнение. Каждая операция принимает на вход пару многокорневых бинарных решающих диаграмм, выполняет нужные манипуляции и возвращает новую многокорневую бинарную решающую диаграмму, представляющую сконструированный в результате объект. С целью ускорения работы алгоритмов именно на этой стадии имеет смысл перенести основную работу из C++-кода в более простой и эффективный C.

На данный момент в библиотеку входит реализация операций с целыми числами на множествах $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$

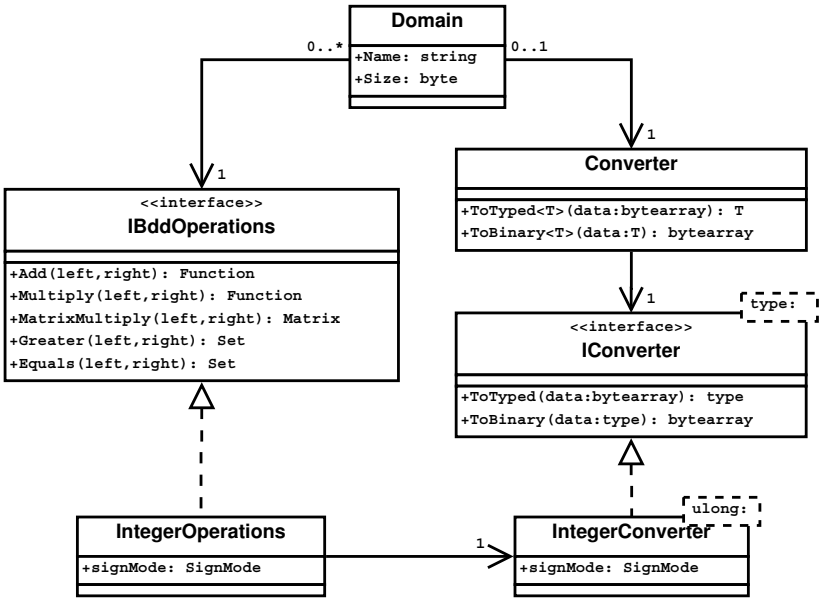


Рис. 2. Реализация основных алгоритмов

(знаковые целые) и $\{0, \ldots, 2^n - 1\}$ (беззнаковые целые). Получить доступ к встроенной реализации можно посредством класса `IntegerOperations` и его статического метода `GetInstance`:

```
// Регистрируем домен "знаковый байт"
Domain byte = manager.RegisterDomain(
    "byte", 8, IntegerOperations::GetInstance());

// Регистрируем домен "беззнаковый байт"
Domain ubyte = manager.RegisterDomain(
    "ubyte", 8,
    IntegerOperations::GetInstance(Unsigned));
```

1.2. Основные алгоритмы

Целочисленной функцией назовем функцию вида: $f : Z_n \to Z_n$[5]. Базовыми операциями над целочисленными функциями, чаще всего применяемыми на практике, являются:

— сложение $s(x) = f(x) + g(x)$; — умножение $p(x) = f(x) * g(x)$; — суперпозиция $c(x) = f(g(x))$; — сравнение $\{x \in Z_n | f(x) > g(x)\}$; — матричное умножение.

Далее предлагаются алгоритмы, реализующие эти операции над функциями, представленными в виде многокорневых бинарных решающих диаграмм[6].

1.2.1. Сумма функций

Пусть функция f представлена набором булевых функций $\{f_i\}_{i=1}^n$, а функция g — набором $\{g_i\}_{i=1}^n$. В этом случае представление $\{s_i\}_{i=1}^n$ функции $s(x) = f(x) + g(x)$ можно получить с помощью следующего рекурсивного алгоритма:

$$
\begin{aligned}
&add_0 = \emptyset,\\
&\forall i \in \{1, \ldots, n\} : s_i = f_i \text{ XOR } g_i \text{ XOR } add_{i-1},\\
&\forall i \in \{1, \ldots, n\} : add_i = \text{ITE}(f_i,\ g_i \vee add_{i-1},\ g_i \wedge add_{i-1}),\\
&s_{n+1} = add_n.
\end{aligned}
$$

Функция $s_i(x)$, представляющая i-ый бит значения $s(x)$, вычисляется как «исключающее или» от трех булевых функций: f_i, g_i и add_{i-1}, где первые две представляют i-е биты суммируемых функций, а add_{i-1} представляет значения, перенесенные с младших разрядов (в случае если при вычислении младших битов произошло переполнение). При этом функция $add_i(x)$, соответствующая значениям, требующим переноса в старший разряд, вычисляется как объединение всех значений, для которых истинны не менее чем две

[5] Здесь и далее для простоты изложения используются функции одной переменной. Обобщение для случая нескольких переменных строится очевидным образом.

[6] Алгоритмы вычисления суммы и произведения функций также представлены в работе [1].

из следующих трех функций: $f_i(x)$, $g_i(x)$ и $add_{i-1}(x)$. Для вычисления add_i используется тернарная функция $if\text{-}then\text{-}else$ (ITE).

Заметим, что после выполнения алгоритма остается множество add_n, соответствующее разряду, который не был представлен в функциях $f(x)$ и $g(x)$. В зависимости от конкретной задачи можно выбрать различные способы обработки этого множества:

- автоматически расширять представление функции $s(x)$ дополнительным разрядом;
- отбросить лишний разряд;
- контролировать область допустимых значений функции.

В первом случае не происходит потери информации, однако его применение может привести к неконтролируемому росту потребности алгоритма в памяти. При применении второго подхода функции фактически рассматриваются на кольце по модулю 2^n, что соответствует привычной для большинства современных микропроцессоров интерпретации. Третий подход позволяет отслеживать переполнения, однако при большом количестве операций может привести к вырождению функций за счет сужения области определения. По нашему мнению, для большинства задач наиболее подходящим оказывается применение второго подхода, однако первый и третий подходы также могут иметь свои специфические области применения.

1.2.2. Произведение функций

Для вычисления функции $p(x) = f(x) * g(x)$ можно воспользоваться следующей формулой:

$$p(x) = \sum_{1}^{n} p^i(x) * 2^{i-1},$$

где n есть количество битов в используемой кодировке, операция умножения на 2^{i-1} реализована как битовый сдвиг, а представление $\{p^i_j\}^n_{j=1}$ функций $p^i(x)$ получено следующим образом:

$$\forall j \in \{1, \ldots, n\} : p^i_j = f_j \wedge g_i.$$

В основу этого алгоритма положены правила побитового вычисления произведения чисел, модифицированные для работы с множествами. Наглядной аналогией алгоритма может служить вычисление произведения чисел «столбиком».

Также как и в случае со сложением, представление функции $p(x) = f(x) * g(x)$ требует большего количества бит — в два раза больше, чем количество бит в представлении $f(x)$ и $g(x)$. Способы обработки дополнительных битов здесь те же: их можно динамически добавить, отбросить или ввести область допустимых значений.

1.2.3. Суперпозиция функций

Одним из важных преимуществ многокорневых бинарных решающих диаграмм является простота вычисления суперпозиции функций. Это преимущество обеспечивается тем фактом, что бинарные решающие диаграммы представляют функции булевых *аргументов*, в то же время многокорневые диаграммы представляют набор функций с булевыми *значениями*, а многотерминальные диаграммы — единую функцию с конечным множеством значений, которое, как правило, шире чем $\{0, 1\}$. В результате, в большинстве существующих пакетов решающих диаграмм, суперпозиция многотерминальных диаграмм не реализована. Например, в пакете CUDD многотерминальные диаграммы можно использовать в качестве аргументов суперпозиции только в том случае, если они принимают значения $\{0, 1\}$.

Пусть функция f представлена набором булевых функций $\{f_i\}_{i=1}^n$, а функция g — набором $\{g_i\}_{i=1}^n$. В этом случае представление $\{c_i\}_{i=1}^n$ функции $c(x) = f(g(x))$ вычисляется следующим образом:

$$\forall i \in \{1, \ldots, n\} : c_i(x_1, \ldots, x_n) = f_i(g_1(x_1, \ldots, x_n), \ldots, g_n(x_1, \ldots, x_n)),$$

где $(x_1, \ldots, x_n) \in \{0, 1\}^n$ есть описанная ранее стандартная битовая кодировка числа $x \in Z_n$.

Заметим, что при вычислении суперпозиции, в отличие от вычисления суммы или произведения функций, не может возникнуть переполнения, так как множество возможных значений функции

$c(x) = f(g(x))$, очевидно, есть нестрогое подмножество множества возможных значений функции $f(x)$.

1.2.4. Сравнение функций

Для определения кусочно-заданных функций в библиотеку введено понятие множества, а также тернарная операция *if-then-else* (см. *разд. 1.1.*). Основным средством описания самих множеств является сравнение значений функций, например:

$$[f > g] \doteq \{x \in Z_n | f(x) > g(x)\} \quad \text{и} \quad [f = g] \doteq \{x \in Z_n | f(x) = g(x)\}^7.$$

Заметим, что множества $[f > g] \subseteq Z_n$ и $[f = g] \subseteq Z_n$ эквивалентны функциям вида $[f > g] : Z_n \to \{0, 1\}$ и $[f = g] : Z_n \to \{0, 1\}$, возвращающим 1 для тех точек, которые входят в искомое множество, и 0 иначе.

Представление множества $[f = g]$ достаточно просто строится по формуле:

$$[f = g] = \neg \left(\bigvee_{i=1}^{n} (f_i \text{ XOR } g_i) \right).$$

Таким образом, множество значений, для которых функции совпадают, определяется как пересечение множеств, на которых совпадают соответствующие биты в представлении функции.

Вычисление множества $[f > g]$ для беззнаковых целых чисел можно провести с помощью следующего рекурсивного алгоритма:

$$\begin{aligned}
&[f > g]_1 = f_1 \wedge \neg g_1, \\
&equalsZone_1 = \neg(f_1 \text{ XOR } g_1), \\
&\forall i \in \{2, \ldots, n\} : [f > g]_i = equalsZone_{i-1} \wedge f_i \wedge \neg g_i, \\
&\forall i \in \{2, \ldots, n\} : equalsZone_i = equalsZone_{i-1} \wedge \neg(f_i \text{ XOR } g_i), \\
&[f > g] = \bigvee_{i=1}^{n} [f > g]_i.
\end{aligned}$$

[7] При этом остальные операции сравнения ($[f >= g]$, $[f < g]$ и т. д.) очевидным образом выражаются через две базовые операции $[f = g]$ и $[f > g]$.

На каждом шаге алгоритма происходит сравнение двух соответствующих битов в представлении функции, а в результирующее множество включаются те значения, для которых соответствующий бит f_i равен 1, а g_i равен 0, при этом все предыдущие (более старшие биты) совпадают (что контролируется посредством дополнительной переменной $equalsZone_{i-1}$). Для того чтобы применить алгоритм сравнения к работе со знаковыми числами, необходимо изменить правило обработки первого бита на $[f > g]_1 = \neg f_1 \vee g_1$ (при этом предполагается, что отрицательные числа представлены в дополнительном коде [8, с. 16]).

1.2.5. Операции над матрицами

В некоторых задачах, например, при проведении верификации систем по спецификации в нечеткой логике (вероятностной верификации) [14], одной из основных операций становится умножение конечнозначных матриц. Заметим, что конечнозначную матрицу $A \in S^{n\times n}$ можно рассматривать как функцию двух переменных $f_A : \{1,\dots,n\}^2 \to S$, определяемую следующим образом: $f_A(i,j) \doteq A_{ij}$. Сумма матриц, таким образом, тривиальным образом реализуется через суммирование соответствующих функций.

Произведение же матриц по своей сути достаточно сильно отличается от произведения двух функций тем, что помимо собственно произведения элементов матриц необходимо суммировать результат по столбцам первой матрицы и строкам второй матрицы:

$$A_{n\times m} * B_{m\times l} = \sum_{k=1}^{m} f_A(i,k) * f_B(k,j),$$

где f_A и f_B есть функции, представляющие матрицы A и B соответственно, $i \in \{1,\dots,n\}$ — переменная, кодирующая строки матрицы A, $k \in \{1,\dots,m\}$ — переменная, кодирующая столбцы A и строки B, а $j \in \{1,\dots,l\}$ — переменная, кодирующая столбцы матрицы B. Далее эту операцию с функциями будем назвать *произведение относительно переменной* k.

Реализовать подобную операцию суммирования в виде набора побитовых операций представляется затруднительным, однако можно предложить рекурсивный алгоритм, управляемый

структурой решающей диаграммы. Пусть v^A и v^B есть вершины решающих диаграмм, $summBy$ — множество индексов переменных, по которым необходимо вычислить сумму, $topIndex$ — базовый индекс, соответствующий индексу наибольшей обработанной к данному моменту переменной. Тогда определим функцию $mmStep(v^A, v^B, summBy, topIndex)$ следующим образом.

1. Если одна из вершин v^A или v^B есть терминальная вершина, представляющая 0, вернуть $f(x_0, \ldots, x_n) = 0$.
2. Если обе вершины v^A и v^B есть терминальные вершины, вернуть $f(x_0, \ldots, x_n) = 1 * 2^s$, где $s = count(summBy \cap (topIndex, n))$ — количество переменных суммирования, индексы которых находятся в интервале $(topIndex, n)$.
3. Положить $i = max(index(v^A), index(v^B))$.
4. Если $index(v^A) = i$, то положить $vl^A = v^A_{low}$, $vh^A = v^A_{high}$, иначе положить $vl^A = vh^A = v^A$.
5. Если $index(v^B) = i$, то положить $vl^B = v^B_{low}$, $vh^B = v^B_{high}$, иначе положить $vl^B = vh^B = v^B$.
6. Если $i \in summBy$, то вернуть $f(x_0, \ldots, x_n) = 2^s * (mmStep(vh^A, vh^B, summBy, i) + mmStep(vl^A, vl^B, summBy, i))$, где $s = count(summBy \cap (topIndex, i))$.
7. Если $i \notin summBy$, то вернуть:
$$f(x_0, \ldots, x_n) = 2^s * \begin{cases} mmStep(vh^A, vh^B, summBy, i), \text{iff } x_i = 1 \\ mmStep(vl^A, vl^B, summBy, i), \text{iff } x_i = 0 \end{cases},$$
где $s = count(summBy \cap (topIndex, i))$.

Если v^A и v^B есть диаграммы, представляющие матрицы из элементов $\{0, 1\}$, то результатом работы алгоритма $mmStep(v^A, v^B, summBy, 0)$ является функция, эквивалентная произведению этих матриц относительно переменных, индексы которых помещены в множество $summBy$. Таким образом, функция $mmStep$ может быть использована для умножения матриц вида $A^h \in \{0, 1\}^{n \times m}$, при этом матрица $A \in \{0, \ldots, 2^p - 1\}^{n \times m}$ строится из таких матриц по стандартной формуле: $A = \sum_{o=0}^{p-1} 2^o * A^o$. В результате, итоговый алгоритм произведения матриц можно описать

следующим образом:

$$A * B = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(2^{i+j-2} * mmStep\left(A_i, B_j, summBy, 0\right)\right),$$

где A_i и B_j есть представления i-го и j-го битов матриц A и B соответственно, а $summBy$ — множество индексов переменных, относительно которых необходимо выполнить умножение (как правило, это переменные, кодирующие столбцы первой и строки второй матриц).

Предложенный алгоритм является модификацией алгоритма, описанного в [4]. Основным отличием является то, что рекурсивная процедура $mmStep$ возвращает многокорневую, а не многотерминальную диаграмму. С точки зрения практической реализации для обоих алгоритмов необходима реализация эффективного кэширования ранее полученных результатов. Это обусловлено тем, что в процессе рекурсии один и тот же участок структуры может быть обработан несколько раз и результаты предыдущего вычисления можно использовать повторно. К сожалению, существующие реализации кэширования ориентированы на хранение однокорневых решающих диаграмм и требуют модификации для работы с многокорневыми диаграммами. В связи с этим в данной работе не приводятся представительные экспериментальные данные для сравнения предложенного алгоритма произведения матриц с классическим.

2. Экспериментальные данные

Так как теоретически оценить размеры решающих диаграмм затруднительно в связи со сложностью оценки повторного использования фрагментов структуры диаграммы, в данной работе мы приводим результаты экспериментального сравнения времени построения и размера представления некоторых стандартных целочисленных функций. Результаты сравнения приведены в таблице, где размер указан в количестве узлов, время указано в миллисекундах, и для каждой функции приведены результаты при различных размерах аргумента в битах.

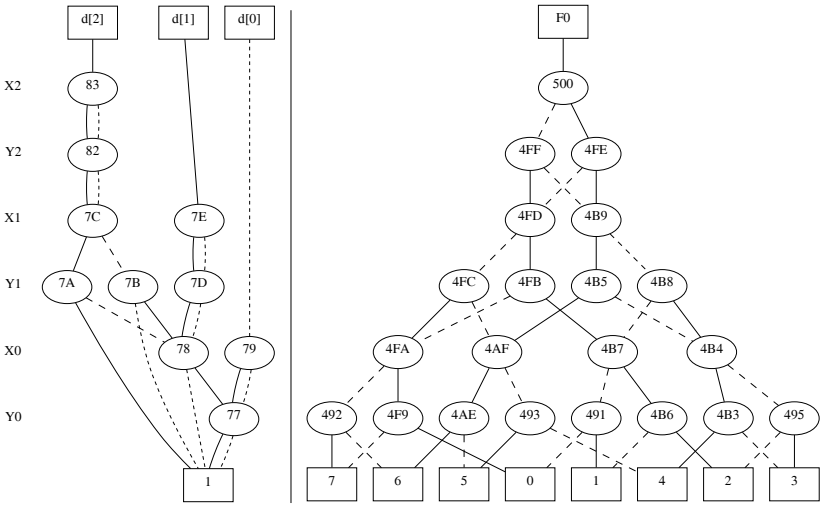
Размер и время построения функций

Функция		Размер		Время построения	
Формула	n	$MTBDD$	$MRBDD$	$MTBDD$	$MRBDD$
$f(x) = x + c$	8	511	9	0	0
	16	131 071	17	78	0
	24	33 554 431	25	63 071	0
	32	—	33	—	0
$f(x, y) = x + y$	8	1021	36	15	0
	16	262 141	76	550 024	0
	24	—	116	—	0
	32	—	156	—	16
$f(x) = x^2$	8	171	55	0	0
	16	43 691	3587	47	62
	24	11 184 811	230 685	55 583	12 480
	32	—	14 953 397	—	1 943 510
$f(x) = c * x$	8	511	46	0	0
	16	131 071	780	62	0
	24	33 554 431	12 330	54 959	47
	32	—	196 696	—	2137
$f(x, y) = x * y$	4	228	75	0	0
	8	54 868	5691	16	16
	12	13 985 108	487 095	16 910	921
	16	—	41 436 807	—	146 968

Для функций с аддитивной структурой ($f(x) = x + c$, $f(x, y) = x + y$) показано существенное преимущество многокорневого подхода: размер представления функции растет линейно относительно размера аргумента в многокорневом случае и экспоненциально в многотерминальном случае. Например, при размере аргументов в 16 бит представление функции $f(x, y) = x + y$ в виде $MRBDD$ состоит из 76 узлов, тогда как аналогичная $MTBDD$ состоит из 262 141 узла. В связи с линейным ростом размера диаграммы применение $MRBDD$ позволяет представлять системы с аддитивной внутренней структурой практически неограниченного размера, тогда как применение $MTBDD$ для аналогичных систем существенно ограничено.

На рис. 3 приведен внешний вид решающих диаграмм, представляющих функцию $f(x, y) = x + y$ при размере аргументов и результата в 3 бита. На рис. 3, *б* изображена структура соответствующей многотерминальной решающей диаграммы, где сплош-

ные линии соответствуют истинным значениям аргументов (v_{high}), а пунктирные — ложным (v_{low}). Из диаграммы наглядно видно, как количество узлов удваивается через каждые два слоя, что и дает экспоненциальный рост диаграммы. На рис. 3, *а* изображена соответствующая многокорневая диаграмма, где сплошные и пунктирные линии имеют тот же смысл, а линии с малым пунктиром соответствуют так называемым *дополнительным* дугам $\bar{v}_{low}$ при прохождении которых значения инвертируются: $f_{\bar{v}_{low}} \doteq \neg f_{v_{low}}$[8]. Рис. 3 наглядно демонстрирует возникновение перекрестных ссылок между подграфами, благодаря чему и обеспечивается линейная скорость роста размера диаграммы.



а *б*

Рис. 3. Структура многокорневой (*а*) и многотерминальной (*б*) решающих диаграмм для функции $f(x, y) = x + y$

Для функций с мультипликативной структурой ($f(x) = x * c$, $f(x) = x^2$, $f(x, y) = x * y$) оба подхода показали экспоненциаль-

[8] Дополнительные дуги позволяют обойтись одной терминальной вершиной и упрощают реализацию операции $\neg$. Более подробно работа с дополнительными дугами изложена в [15].

ный рост размера представления, однако во всех случаях основание экспоненты для многокорневых диаграмм меньше, чем для многотерминальных. Например, для наиболее «тяжелой» функции $f(x, y) = x * y$ скорость роста $MTBDD$ пропорциональна 4^n, тогда как для $MRBDD$ — 3^n, что для аргументов размером 12 бит дает 13 985 108 элементов в $MTBDD$ или 487 095 элементов в $MRBDD$. Для функции $f(x) = x^2$ скорость роста $MTBDD$ пропорциональна 2^n, а в случае $MRBDD$ — $1,7^n$. Для функции $f(x) = x * c$ (где c есть некоторая константа) показатели скорости роста составили 2^n для $MTBDD$ и $1,4^n$ для $MRBDD$[9].

Заключение

В данной работе предлагаются новые структуры данных для представления целочисленных функций и матриц: многокорневые бинарные решающие диаграммы ($MRBDD$), а также алгоритмы выполнения некоторых стандартных операций над целочисленными функциями и матрицами в таком представлении. За счет более эффективного повторного использования элементов структуры многокорневые бинарные решающие диаграммы оказываются более компактной формой представления по сравнению с широко распространенными многотерминальными бинарными решающими диаграммами ($MTBDD$), что подтверждается экспериментальными результатами.

Для представления функций и систем с аддитивной структурой показано значительное снижение необходимого объема памяти и скорости его роста (линейный рост $MRBDD$ против экспоненциального роста $MTBDD$). Для функций и систем с мультипликативной структурой выигрыш в производительности не столь существен, но заметен: скорость роста $MRBDD$, как и у $MTBDD$, экспоненциальна, однако основание экспоненты меньше.

Экспериментальные результаты показывают, что многокорневые бинарные решающие диаграммы являются перспективной заменой многотерминальных бинарных решающих диаграмм, в том

[9]Скорость роста $MRBDD$ представления в данном случае зависела от выбора c. В таблице приведены показатели для наиболее быстрорастущего случая $c = 2^{n/2} - 1$.

числе и в таких задачах, как верификация по спецификации в нечеткой логике [14], представление и выполнение операций над распределениями вероятности [3], анализ сетей Петри [11] и других моделей вычислительных систем [12].

Для расширения области применения предложенного подхода предполагается реализовать эффективный механизм кэширования многокорневых решающих диаграмм, что позволит эффективно реализовать алгоритм умножения матриц. Кроме того, важным потенциальным направлением развития данного подхода является разработка его обобщения для представления и выполнение операций над вещественными функциями и матрицами.

Список литературы

[1] *Бугайченко Д. Ю.* Операции над целочисленными функциями, представленными в виде набора бинарных разрешающих диаграмм // Информационные технологии моделирования и управления. 2009. № 3. С. 358–365.

[2] *Карпов Ю. Г.* Model Checking. Верификация параллельных и распределенных программных систем. БХВ-Петербург, 2010. 552 с.

[3] *Afshin A.* Probabilistic Decision Diagrams for Exact Probabilistic Analysis // ICCAD'07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design. Piscataway, NJ, USA: IEEE Press, 2007. P. 266–272.

[4] *Bahar R., Frohm. E, Gaona C., et al.* Algebraic Decision Diagrams and Their Applications // Proc. IEEE/ACM International Conference on CAD. Santa Clara (California): IEEE Computer Society Press, 1993. P. 188–191.

[5] *Bryant R. E.* Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams // ACM Computing Surveys. 1992. Vol. 24, No. 3. P. 293–318.

[6] Code::Blocks C++ IDE. http://www.codeblocks.org/

[7] *Fujita M., McGeer P. C., Yang J. C.-Y.* Multi-Terminal Binary Decision Diagrams: an Efficient Datastructure for Matrix Representation // Form. Methods Syst. Des. 1997. Vol. 10, No. 2-3. P. 149–169.

[8] *Harris D., Harris S.* Digital Design and Computer Architecture. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. 569 p.

[9] *Janssen G.* A Consumer Report on BDD Packages // Integrated Circuit Design and System Design, Symposium on. 2003. P. 217.

[10] *Jensen R. M.* Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains: Ph.D. Thesis / Carnegie Mellon University. 2003. June. 221 p.

[11] *Jinqing Y. A., Gianfranco C.* Decision-Diagram-Based Techniques for Bounded Reachability Checking of Asynchronous Systems // International Journal on Software Tools Technol. Transf. 2009. Vol. 11, No. 2. P. 117–131.

[12] *Kanupriya G., Nikhil J.* An Algebraic Decision Diagram (ADD) Based Technique to Find Leakage Histograms of Combinational Designs // ISLPED'05: Proceedings Of The 2005 International Symposium On Low Power Electronics And Design. New York, NY, USA: ACM, 2005. P. 111–114.

[13] *Lind-Nielsen J.* BDD Package BuDDy. http://buddy.sourceforge.net

[14] *Parker D.* Implementation of Symbolic Model Checking for Probabilistic System: Ph.D. Thesis / University of Birmingham. 2002. 222 p.

[15] *Somenzi F.* CUDD: Colorado University Decision Diagram Package. http://vlsi.colorado.edu/ fabio/CUDD

[16] *Yang T.* Computational Verb Decision Trees // International Journal of Computational Cognition. 2006. Vol. 4, No. 4. P. 34–46. http://www.ijcc.us