Санкт-Петербургский Государственный
Университет
Математико-механический факультет
Кафедра системного программирования

Применение диаграмм двоичных решений для задач анализа потока данных

Дипломная работа студента 544-й группы
Ловцюса Андрея Вячеславовича

Научный руководитель	..... (Друнин А.В.)
Рецензент, к.ф-м.н	..... (Булычев Д.Ю.)
“Допустить к защите”, заведующий кафедрой, профессор, д.ф-м.н.	..... (Терехов А.Н.)

Санкт-Петербург
2007

Содержание

1 Введение

Анализы достигающих определений, доступных выражений, живых переменных являются классическими задачами анализа потока данных. Традиционный способ их решения [1] достаточно эффективен для программ среднего и малого размеров. Однако, при анализе программ с очень большим количеством операторов и переменных, использование этого метода зачастую приводит к большим затратам памяти и времени. Это объясняется тем, что битовые вектора, используемые для представления множеств, характеризующих поток данных в программе, становятся слишком большими, а их вычисление медленным.

Одним из вариантов улучшения метода решения этих задач является замена представления информации о потоке. От нового представления требуется компактность, а от операций над этим – представлением быстрота.

Диаграммы двоичных решений (Binary Decision Diagrams, BDD), изобретенные Ли [2] и Акерсом [3], являются структурой данных, позволяющей представлять булевы функции. Предложенные Браянтом [4] разновидности BDD сделали это представление более компактным и эффективным.

Любое конечное множество можно представить с помощью BDD, используя двоичное кодирование его элементов. При этом операции над множествами сводятся к операциям над булевыми функциями. Таким образом, диаграммы двоичных решений позволяют компактно представлять множества большого размера, что и дает основу их применению при решении потоковых задач.

Данная дипломная работа ставит своей целью применение диаграмм двоичных решений для эффективного решения задачи достигающих определений.

В главе 2 приведено описание диаграмм двоичных решений и распространенных способов их использования. Глава 3 содержит описание классического способа решения задачи достигающих определений и его недостатки. Приводится описание способов решения потоковых задач с использованием BDD. В главе 4 обсуждаются способы сокращения размеров диаграмм при использовании их в методах описанных в предыдущей главе. Приводится сравнение производительности новых решений и традиционного метода.

2 Обзор существующих подходов

Одной из основных проблем анализа программ является необходимость представлять и оперировать наборами больших множеств. Диаграммы двоичных решений – это структура данных, которая широко используется в верификации моделей (Model checking) для компактного представления больших множеств состояний. Но в общем случае, верификация моделей и анализ программ значительно отличаются по формам оперируемых данных. Верификатор моделей использует BDD для вычисления достигаемого множества состояний конечного автомата. Анализ программ, в свою очередь, оперирует более широким многообразием типов данных, для которых не очевидно как именно закодировать их в BDD и как ими оперировать. Тем не менее, некоторые задачи программного анализа могут быть решены с применением диаграмм двоичных решений.

2.1 BDD

Определение 1 *Диаграмма двоичных решений (Binary Decision Diagram, BDD) – это направленный, ацикличный, корневой граф такой, что:*

- *Множество вершин содержит две терминальные вершины, помеченные 1 и 0. Эти вершины не имеют исходящих дуг;*
- *Каждая нетерминальная вершина v имеет метку var(v) и две исходящие дуги: low(v) и high(v).*

Диаграмма двоичных решений задает булеву функцию, если ее вершины помечены переменными этой функции. Для данных значений переменных значение функции определяется проходом по пути из вершины в терминальную вершину. Путь проходит через дугу low(v) если значение var(v) ложно и по дуге high(v) если оно истинно. Значение функции определяется меткой терминальной вершины конца пути [5].

Для эффективного хранения и манипуляций с данными на диаграммы двоичных решений накладываются некоторые ограничения. На множестве переменных X задается отношение порядка $<$.

Определение 2 *Упорядоченной диаграммой двоичных решений (Ordered Binary Decision Diagram, OBDD) называется такая BDD, что для любой нетерминальной вершины v_1 и любого её нетерминального потомка v_2 выполняется $var(v_1) < var(v_2)$.*

Основной причиной эффективности BDD стало появление правил сокращения OBDD и введение сокращенных упорядоченных диаграмм двоичных решений.

Определение 3 *Сокращенной упорядоченной диаграммой двоичных решений (Reduced Ordered Binary DecisionDiagram, ROBDD) называется такая OBDD, что для любых нетерминальных вершин* v_1 *и* v_2

- $low(v_1) \neq high(v_1)$
- *если* $var(v_1) = var(v_2) \wedge low(v_1) = low(v_2) \wedge high(v_1) = high(v_2)$, *то* $v_1 = v_2$

Представление булевой функции с помощью ROBDD является каноническим – для фиксированного порядка переменных, любые две ROBDD для одной функции изоморфны. Это свойство имеет важные следствия. Так, эквивалентность функций может быть легко проверена. Функция имеет истинные значения если и только если её ROBDD представление не соответствует единственной терминальной вершине с меткой 0. Любая тавтологичная функция должна иметь терминальную вершину с меткой 1 в качестве своего ROBDD представления. Если функция не зависит от переменной x, то её ROBDD представление не содержит вершин помеченных x.

2.2 Влияние упорядоченности переменных BDD

Форма и размер BDD[1]-представления Булевой функции зависит от порядка на множестве переменных. Так на рисунке 1 приведены два представления функции $(a \vee b) \wedge (c \vee d)$. На левой BDD переменные отсортированы a $<$ b $<$ c $<$ d, тогда как на правой a $<$ c $<$ b $<$ d. Если количество переменных функции достаточно велико, то удачный выбор порядка переменных может серьёзно сказаться на размере BDD. Известно, что задача нахождения оптимального порядка даже для единственной диаграммы двоичных решений NP-полна [6].

В большинстве случаев применения BDD выбирается один фиксированный порядок переменных и все диаграммы строятся в соответствии с этим порядком. Этот порядок либо выбирается фиксировано для решаемой задачи или же с помощью эвристического анализа конкретной системы, представляемой с помощью BDD. Все эвристические методы определения порядка можно разделить на статические и динамические [7]. Статические подходы опираются

[1]Здесь и далее под BDD подразумевается ROBDD.

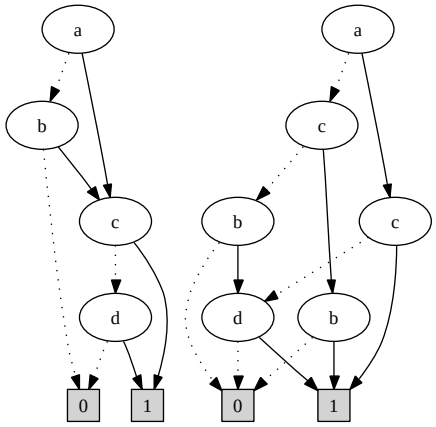

Рис. 1: Представления функции $(a \vee b) \wedge (c \vee d)$ с разными порядками переменных.

на различные свойства, собранные во время структурного анализа кодируемой системы. По результатам анализа определяется наиболее оптимальный порядок переменных. Динамические подходы используют какой-либо начальный порядок, который затем анализируется и меняется так, чтобы стоимостная функция (часто это размер BDD) уменьшилась. Методы, наиболее сильно сокращающие размер диаграммы, зачастую мало эффективны по времени.

2.3 Представление математических абстракций с помощью BDD

Некоторые приложения использующие BDD нуждаются в представлении именно булевых функций и операциях над ними. Однако сила символьных Булевых вычислений происходит от возможности двоичных значений и булевых операций представлять различные математические системы.

Ключ к успешному использованию сильных сторон символьных булевых вычислений заключается в необходимости переформулировать задачу в форме, в которой все оперируемые объекты будут представлены с помощью булевых функций.

2.3.1 Представление множеств

Рассмотрим конечное множество A с количеством элементов $|A| = N$. Можно закодировать элементы из A с помощью двоичных векторов длиной n, где $n = \lceil log_2 N \rceil$. Кодирование элементов обозначается

функцией $\sigma : A \to \{0,1\}^n$, которая отображает каждый элемент множества A в конкретный вектор из n бит. С помощью $\sigma_i(a)$ обозначается i-й элемент вектора. Тогда функцию отображающую множество A на само себя, $f : A \to A$ можно представить как вектор из n Булевых функций $\vec{f}$, где каждая $f_i : \{0,1\}^n \to \{0,1\}$ определяется как $f_i(\sigma(a)) = \sigma_i(f(a))$.

Имея кодировку множества A мы можем представлять его подмножества с помощью характеристических функций. Множество $S \subseteq A$ представляется Булевой функцией $\chi_S : \{0,1\}^n \to \{0,1\}$, где

$$\chi_S(\vec{x}) = \sum_{a \in S} \prod_{1 \le i \le n} x_i \bar{\oplus} \sigma_i(a),$$

где $\bar{\oplus}$ обозначает операцию исключающее или. Тогда операции над множествами могут быть представлены с помощью Булевских операций над их характеристическими функциями.

$$\chi_\emptyset = 0$$

$$\chi_{S \cup T} = \chi_S + \chi_T$$

$$\chi_{S \cap T} = \chi_S \cdot \chi_T$$

$$\chi_{S - T} = \chi_S \cdot \bar{\chi_T}$$

Множество S является подмножеством множества T если и только если $\chi_S \cdot \bar{\chi_T} = 0$.

2.3.2 Представление отношений

Отношение размерности k может быть определено как множество упорядоченных наборов размера k. Следовательно мы можем представлять отношения с помощью характеристических функций. Рассмотрим бинарное отношение $R \subseteq A \times A$. Этому отношению соответствует Булева характеристическая функция χ_R:

$$\chi_R(\vec{x}, \vec{y}) = \sum_{a \in A} \sum_{b \in A, aRb} \left[\prod_{1 \le i \le n} x_i \bar{\oplus} \sigma_i(a) \right] \cdot \left[\prod_{1 \le i \le n} y_i \bar{\oplus} \sigma_i(b) \right].$$

С этим представлением мы можем осуществлять такие операции как пересечение, объединение и разность отношений, применяя булевские операции над их характеристическими функциями.

Мы также можем вычислить композицию отношений:

$$\chi_{R \circ S}(\vec{x}, \vec{y}) = \exists \vec{z} \left[\chi_R(\vec{x}, \vec{z}) \cdot \chi_S(\vec{z}, \vec{y}) \right],$$

где $R \circ S$ обозначает композицию отношений R и S.

Вычисляя композицию последовательно, мы можем получить транзитивное замыкание отношения. Функция χ_{R*} вычисляется как предел последовательности функций χ_{R_i}, каждая из которых определяет отношение:

$$R_0 = I$$

$$R_{i+1} = I \cup R \circ R_i$$

где I обозначает тождественное отношение. Вычисление завершается по достижении итерации i, для которой $\chi_{R_i} = \chi_{R_{i-1}}$. При этом проверка эквивалентности осуществляется эффективно за счет представления отношений в виде BDD.

2.4 BDD в Points-to анализе

Одна из областей программного анализа, где диаграммы двоичных решений могут быть эффективно применены, это анализ указателей. Так в работе [8] авторы привели способ проведения анализа указателей, основанного на подмножествах, для языка Java. Эта задача может быть рассмотрена как задача нахождения областей памяти, на которые может указывать переменная в программе. Основная проблема в разработке эффективных анализов указателей, основанных на подмножествах, состоит в том, что для больших программ количество и размер points-to множеств могут быть очень большими. В этих условиях диаграммы двоичных решений оказались очень эффективными с точки зрения времени работы, расхода памяти и удобства использования.

Рассмотрим способ применения BDD на простом примере:

```
A: a = new O();
B: b = new O();
C: c = new O();
a = b;
b = a;
c = b;
```

Тогда points-to отношение для этой программы: $\{(a, A), (a, B), (b, A), (b, B), (c, A), (c, B), (c, C)\}$, где пара (a, A) говорит о том что переменная a может указывать на объект созданный в точке программы A. Используя 00 для того, чтобы обозначить a и A, 01 для b и B, 10 для c и C, мы можем представить points-to отношение как множество $\{0000, 0001, 0100, 0101, 1000, 1001, 1010\}$. На рисунке 2 приведены диаграммы двоичных решений представляющие это множество, где переменные a, b и c закодированы при помощи

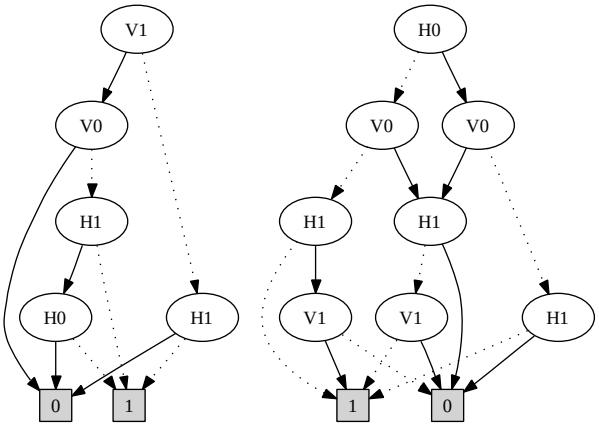

Рис. 2: BDD соответсвующие отношению $\{(a,A),(a,B),(b,A),(b,B),(c,A),(c,B),(c,C)\}$

переменных $V1$ и $V0$, а точки создания объектов A, B и C при помощи переменных $H1$ и $H0$. На левой BDD переменные отсортированы $V1 < V0 < H1 < H0$, а на правой $H0 < V0 < H1 < V1$.

Одной из важных операций используемых при вычислении points-to отношения является операции композиции отношений. Изначальное points-to отношение для приведенного примера $\{(a,A),(b,B),(c,C)\}$ (первые три строки кода) и множество дуг образуемых присваиваниями $\{(b,a),(a,b),(b,c)\}$(последние три строки кода). Пара (a,b) соответствует присваиванию b:=a. Для представления этого отношения выделяются ещё две переменные BDD. Таким образом вместо переменных $V0, V1$ мы получаем переменные $V10, V11, V20, V21$.

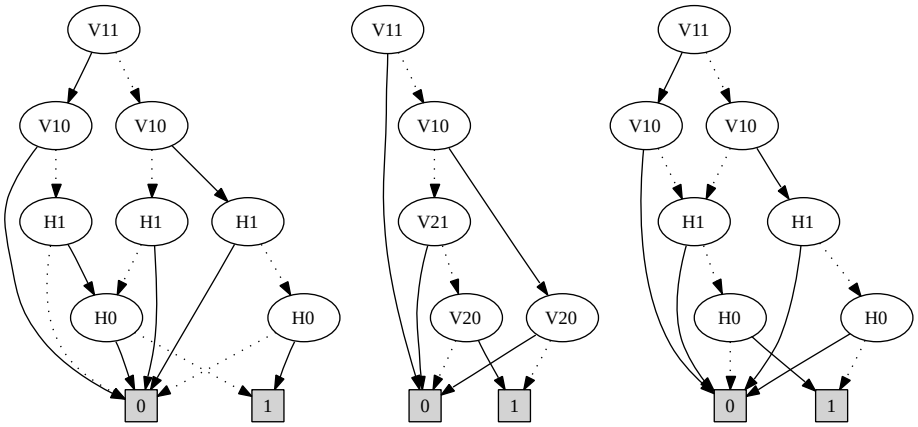

Рис. 3: Композиция отношений

На рисунке 3 левая BDD представляет отношение $\{(a, A), (b, B), (c, C)\}$, средняя – множество дуг $\{(b, a), (a, b), (b, c)\}$, а правая – результат композиции первых двух по отношению к переменным $V10$ и $V11$ и замены переменных $V20$ и $V21$ на $V10$ и $V11$. Таким образом, за счет операции композиции отношений, происходит переход множества значений от указателя в правой части присваивания к указателю в левой части.

3 Анализ достигающих определений с использованием BDD

Определение 4 *Определением называется присваивание какого-либо значения какой-либо переменной.*

Определение 5 *Определение d переменной достигает точку p программы, если существует путь исполнения программы из d в p такой, что переменная в точке p может принимать значение, присвоенное в присваивании d.*

Задачей анализа достигающих определений является нахождение всех определений, которые могут по какому либо пути исполнения достигнуть выбранную точку программы.

3.1 Классический метод решения

Анализ достигающих определений обычно осуществляется в классической форме известной, как итеративная прямая задача на битовых векторах [1, 9]. Итеративной она является потому что строится набор потоковых уравнений для представления потока информации, которые решаются с помощью итераций, начиная с некоторого набора начальных значений. Прямой она является потому что поток информации осуществляется по направлению дуг потока управления. Основанная на битовых векторах потому что каждое определений может быть представлено единицей (определение достигает точку) или нулём (не достигает) и поэтому набор достигающих определений может быть представлен битовым вектором.

Информация о том, как влияет каждый узел графа потока управления программы на достижимость определений, фиксируется посредством двух множеств (битовых векторов), обозначаемых $KILL$ и GEN. Множество $GEN\,(i)$ содержит те определения, которые генерируются в вершине i. Множество $KILL\,(i)$ содержит те определения, которые перетираются в вершине i. Для хранения информации о том, какие определения достигают вершину и какие из них не перетираются в вершине, вводятся множества (битовые вектора) IN и OUT.

В начале работы анализа достигающих определений для каждого узла i вектора $IN\,(i)$ и $OUT\,(i)$ инициализируются последовательностями нулей (такая последовательность соответствует пустому множеству).

Потоковые уравнения для вершин графа потока управления для всех i выглядят следующим образом:

$$IN\,(i) = \bigcup_{j \in Pred(i)} OUT\,(j)$$

$$OUT\,(i) = GEN\,(i) \cup (IN\,(i) - KILL\,(i))$$

Для достижения фиксированной точки, являющейся результатом анализа достигающих определений, вектора IN и OUT вычисляются пока для всех узлов вектора IN и OUT не меняются после очередного перевычисления.

Для более эффективной работы алгоритма вместо вершин граф потока управления используют линейные участки, заранее вычисляя для них GEN и $KILL$ множества. Однако использование этого классического метода для анализа больших программ зачастую оказывается неэффективно. Это объясняется тем, что представление множеств с помощью битовых векторов избыточно, их размер никак не зависит от реального количества достигающих определений. Несмотря на то, что операции над битовыми векторами могут быть эффективно реализованы, в случае очень большого размера векторов время выполнения этих операций становится существенным.

3.2 Замена векторов на BDD

В предыдущей главе были описаны диаграммы двоичных решений. Также была описана задача о достигающих определениях и недостатки классического метода её решения. BDD за счёт своей способности эффективно представлять большие множества и отношения, представляется структурой, подходящей для улучшения классического подхода к решению задачи о достигающих определениях, а также аналогичных ей задач. Идея использования диаграмм двоичных решений для решения задач анализа потока данных не является абсолютно новой. Так в работе [10] приводится доказательство того факта, что анализ потока данных является проверкой модели абстрактной интерпретации программы. Проверка моделей – это та область, где BDD получили широкое распространение. В работе [11] диаграммы двоичных решений используются для решения задачи о живых переменных, однако в ней не рассматриваются вопросы эффективности такого использования BDD.

Один из возможных способов улучшить классический способ решения задачи о достигающих определений это изменить представление

множеств IN, OUT, GEN и $KILL$. Поскольку разница между размером битового вектора и количеством достигающих (генерируемых, затираемых) определений для большой программы велика, то использование диаграмм двоичных решений может привести к существенному выигрышу как по количеству расходуемой памяти, так и по времени работы алгоритма.

Предположим, что мощность множества определений A в программе равно N. Будем кодировать все определения последовательностями битов длиной $n = \lceil log_2 N \rceil$, то есть, зададим отображение $\sigma : A \to \{0,1\}^n$. Тогда для каждой вершины графа потока управления все множества IN, OUT, GEN и $KILL$, являющиеся подмножествами множества всех определений, будем представлять диаграммами двоичных решений соответствующими характеристическим функциям этих множеств. То есть для каждой вершины графа потока управления задаются функции $\chi_{IN(i)}$, $\chi_{OUT(i)}$, $\chi_{GEN(i)}$ и $\chi_{KILL(i)}$.

Перед началом работы алгоритма для любого i из множества вершин графа потока управления верно:

$$\chi_{IN(i)} = 0,$$

$$\chi_{OUT(i)} = 0.$$

Этим функциям соответствуют BDD состоящие только из одной терминальной вершины 0

BDD представление любого множества $S = \{a_1, \ldots, a_k\}$ осуществляется следующим образом:

$$\forall j \in \{1, \ldots, k\} : \chi_{\{a_j\}}(\vec{x}) = \prod_{1 \le i \le n} x_i \bar{\oplus} \sigma_i(a),$$

$$\chi_S = \sum_{1 \le j \le k} \chi_{\{a_j\}}.$$

Таким способом заполняются диаграммы двоичных решений для $\chi_{GEN(i)}$ и $\chi_{KILL(i)}$.

Для всех вершин графа потока управления потоковые равенства в терминах Булевых функций представленных с помощью BDD выглядят так:

$$\chi_{IN(i)} = \sum_{j \in Pred(i)} \chi_{OUT(j)},$$

$$\chi_{OUT(i)} = \chi_{GEN(i)} + \left(\chi_{IN(i)} - \chi_{KILL(i)}\right).$$

После инициализации всех диаграмм осуществляется итеративное вычисление решения с использованием рабочего множества workset, по всем вершинам графа потока управления (множество Vertices):

Алгоритм 1

```
for all i∈Vertices
  workset.push(i);
while (¬workset.empty())
{
  i := workset.pop();
  χ_IN(i) := Σ_{j∈Pred(i)} χ_OUT(j);
  χ_Temp := χ_OUT(i);
  χ_OUT(i) := χ_GEN(i) + (χ_IN(i) − χ_KILL(i));
  if (χ_OUT(i) ≠ χ_Temp)
    for all j∈ Succ(i)
      if (j ∉ workset)
        workset.push(j);
}
```

По завершении работы итеративного алгоритма для каждой вершины i функция $\chi_{IN(i)}$ будет являться характеристической функцией множества определений достигающих эту вершину.

Преимуществами данного подхода по отношению к классическому являются сокращение расходов памяти на хранение множеств, содержащих информацию о потоке, а также замена операций над битовыми векторами операциями над BDD, которые более эффективны в некоторых случаях большого размера множества присваиваний программы.

3.3 Использование глобальных BDD

Модификация классического метода решения задачи о достигающих определениях, основанная на замене векторов на BDD, сокращает размеры представления множеств и, возможно, уменьшает время оперирования ими. Однако количество итераций по вершинам графа потока управления остается неизменным. При этом количество итераций может быть очень велико и оно сильно зависит от порядка обработки вершин. Будем использовать глобальные диаграммы для того чтобы избавиться от этих недостатков.

До этого момента мы рассматривали наборы множеств характеризующих поток информации через какую-либо конкретную вершину графа потока управления. Теперь же мы будем рассматривать отношения принадлежности какого-либо определения какой-либо вершине. То есть вместо набора GEN множеств, свойство вершин генерировать определения будет представлено отношением

$GlobalGEN \subseteq V \times A$, где A это множество определений, а V это множество вершин графа потока управления. Если тройка Аналогично вводится отношение $GlobalKILL$. Представим дуги графа потока управления с помощью отношения $Edges \subseteq V \times V$. То есть, если $(a, b) \in Edges$, то в графе потока управления существует дуга с началом в a и концом в b.

Обозначим за n_{def} количество переменных необходимых для кодирования определений и за n_{vert} количество переменных необходимых для кодирования вершин.

Перенумеруем множество вершин представляющих начала дуг, задав отображение $\sigma_{src} : V \to \{0,1\}^{n_{vert}}$. Аналогично зададим отображения σ_{dst} для нумерации коцов дуг и $\sigma_{def} : A \to \{0,1\}^{n_{def}}$ для нумерации определений. Тогда характеристическая функция для $Edges$:

$$\chi_{Edges}\left(\vec{src}, \vec{dst}\right) =$$

$$\sum_{v1 \in V} \sum_{v2 \in V, v1 Edges v2} \left[\prod_{1 \le i \le n_{vert}} src_i \bar{\oplus} \sigma_{src_i}(v1)\right] \cdot \left[\prod_{1 \le i \le n_{vert}} dst_i \bar{\oplus} \sigma_{dst_i}(b)\right],$$

а для $GlobalKILL$:

$$\chi_{GlobalKILL}\left(\vec{src}, \vec{def}\right) =$$

$$\sum_{v1 \in V} \sum_{v2 \in V, v1 Edges v2} \left[\prod_{1 \le i \le n_{vert}} src_i \bar{\oplus} \sigma_{src_i}(v1)\right] \cdot \left[\prod_{1 \le i \le n_{def}} def_i \bar{\oplus} \sigma_{def_i}(b)\right],$$

$GlobalKILL$, $GlobalGen$ и $Edges$ заполняются путем создания BDD для отдельных пар (элементов отношения) и последующим их сложением.

В отличие от предыдущего метода мы отказываемся от использования множеств IN и OUT. Вместо них будем использовать отношение $GlobalSolution \subseteq V \times A$. Наличие в $GlobalSolution$ пары элементов (v, a) будет показывать, что определение a достигает вершину v. В начале работы алгоритма $GlobalSolution$ инициализируется пустым множеством.

Для осуществления симуляции потока информации по дугам графа потока управления будем использовать композицию отношений:

$$\chi_{Edges \circ GlobalSolution}\left(\vec{v}, \vec{d}\right) = \exists \vec{u} \in V \left[\chi_{Edges}(\vec{u}, \vec{v}) \cdot \chi_{GlobalSolution}\left(\vec{u}, \vec{d}\right)\right].$$

Для получения решения будем последовательно перевычислять GlobalSolution следующим образом:

$$\chi_{Tmp} = (\chi_{GlobalSolution} - \chi_{GlobalKill}) \cup \chi_{GlobalGen}$$

$$\chi_{GlobalSolution} = \chi_{Edges \circ Tmp}.$$

Будем проводить итерации перевычисления до тех пор, пока $\chi_{GlobalSolution}$ перестанет изменяться. Проводя вычисления таким образом, мы за одну итерацию получаем данные которые были бы получены предыдущим методом, если бы тот одновременно обработал все вершины графа.

Приведём алгоритм решения задачи о достигающих определениях с помощью глобальных диаграмм:

Алгоритм 2

```
changed := false;
do
{
  χOldSolution := χGlobalSolution;
  χTmp := (χGlobalSolution − χGlobalKill) ∪ χGlobalGen;
  χGlobalSolution := χEdges∘Tmp; //вычисление композиции
  changed := (χGlobalSolution ≠ χOldSolution);
}
while (changed);
```

По завершении работы итеративного алгоритма, $\chi_{GlobalSolution}$ будет являться характеристической функцией отношения достижимости вершин графа потока управления определениями программы. Тогда множество определений достигающий вершину v будет равно $\{d : (v, d) \in GlobalSolution\}$.

Такой подход хорош тем, что вместо большого количества итераций по узлам графа потока управления, алгоритм совершает значительно меньшее количество перевычислений булевых функций. Количество итераций в новом подходе равно максимальной длине пути по которому определение достигает конечной вершины или перетирается другим определением. Это объясняется тем, что за одну итерацию каждое определение продвигается из начала дуги в её конец. Общая эффективность подхода сильно зависит от эффективности операций над характеристическими функциями, то есть над BDD.

3.4 Реализация

Данная дипломная работа осуществлялась в рамках разработки системы автоматического реинжиниринга приложений “Modernization Workbench” [12]. Эта система осуществляет ряд анализов программ, написанных на старых языках программирования. Некоторые из этих анализов используют результаты анализа достигающих определений.

Реализация этого анализа в системе основана на классическом методе, использующем битовые вектора. Для вычисления достигающих определений алгоритму предоставляется набор определений и использований, а также граф потока управления. По завершении итеративной стадии алгоритма должна быть заполнена структура, хранящая соответствие между использованиями и достигающими определениями. Поэтому для методов использующих BDD важна производительность не только стадии итерирования, но и стадии заполнения диаграмм, а также стадии извлечения посчитанных зависимостей.

При реализации методов решения задачи о достигающих определениях использовалась open source библиотека диаграмм двоичных решений BuDDy [13]. Эта библиотека написана на языке C и имеет объектно-ориентированный интерфейс на C++. В ней реализованы основные операции над сокращенными упорядоченными диаграммами двоичных решений. Для достижения высокой производительности в BuDDy используются: общий массив для хранения вершин, кэш операций и автоматическая сборка мусора. Для визуализации диаграмм использовалась функция записи в dot-формат [14].

Важным свойством BuDDy является разделение между всеми диаграммами общих под-диаграмм. Это означает, что BDD представляющие несвязанные друг с другом множества могут разделять вершины, тем самым снижая затраты памяти. Так на рисунке 4 представлены две BDD разделяющие вершины. BDD с корнем в левой

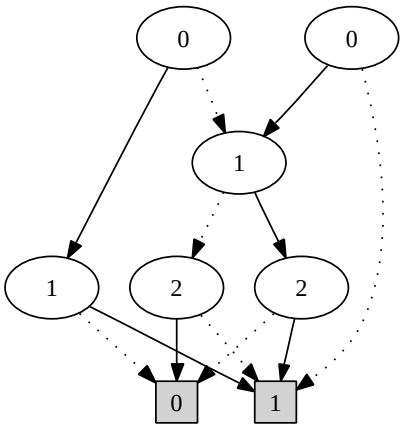

Рис. 4: Представление двух множеств

вершине помеченной нулем соответствует множеству $\{110, 111, 011, 000\}$,

тогда как BDD с корнем в правой вершине соответствует множеству $\{111, 100, 010, 011, 001, 000\}$.

BuDDy предоставляет низкоуровневые операции над BDD, которые оперируют диаграммами и наборами переменных. Для удобства реализации анализа достигающих определений, автором дипломной работы были реализованы классы-оболочки, позволяющие оперировать множествами и отношениями и скрывающие их внутренне представление, основанное на BDD. Для целей сокращения размеров диаграмм были реализованы классы, отвечающие за кодирование элементов множеств двоичными последовательностями и установление соответствия между элементами этих последовательностей и переменных BDD.

4 Минимизация размеров BDD

При решении каких-либо задач с использованием Binary Decision Diagrams очень важно добиться как можно более малого их размера. Большинство операций над BDD имеют полиномиальную сложность. Так, например, операция $APPLY$, с помощью которых реализуются основные операциями над множествами закодированными в BDD, имеет временную сложность $O(m_f m_g)$, где m_f и m_g это количество узлов в BDD-представлении функций операндов f и g [5].

При решении задачи о достигающих определениях способами описанными в предыдущей главе, необходимо определить нумерацию элементов множеств вершин V и присваиваний A, а также составить множества переменных BDD которыми они будут кодироваться.

Число переменных необходимых для кодирования элементов множества A битовыми последовательностями обозначим $n_{def} = \lceil log_2 |A| \rceil$. Для кодирования вершин требуется $n_{vrt} = \lceil log_2 |V| \rceil$ переменных BDD и $2n_{vrt}$ для представления дуг. Тогда количество переменных необходимых для реализации метода использующего глобальные BDD равно $n_{def} + 2n_{vrt}$. Для кодирования элементов множества определений потребуется множество переменных $\left\{d1, \ldots, d_{n_{def}}\right\}$. Для кодирования множества дуг потребуется два множества переменных. Одно для кодирования начал дуг, равное $\{vs1, \ldots, vs_{n_{vrt}}\}$. И другое для кодирования концов дуг, равное $\{vd1, \ldots, vd_{n_{vrt}}\}$. Тогда опираясь на результаты авторов [8], сравнивших разные порядки переменных для схожей задачи, зададим порядок переменных следующим образом: $vs1 < vd1 < vs2 < vd2 < \ldots < vs_{n_{vrt}} < vd_{n_{vrt}} < d1 < \ldots < d_{n_{def}}$.

После установления порядка переменных остается определить кодировку элементов множеств последовательностями битов. Для того чтобы BDD, представляющая некоторое подмножество C множества D, была компактна, необходимо чтобы элементы D входящие в C имели как можно более похожие кодировки. Оптимальной будет такая кодировка, в которой коды элементов множества имеют общий префикс, наибольший из возможных. Довольно компактного представления можно добиться последовательной нумерацией элементов C и кодировании их двоичными представлениями номеров. Но невозможно закодировать элементы множества так, чтобы нумерация любого количества подмножеств была последовательной. Поскольку в случае решения задачи достигающих определений мы имеем дело с большим количеством подмножеств, то необходимо найти компромиссный способ нумерации, который приведет к малым размерам BDD. Так на левой части рисунка 5 изображена BDD

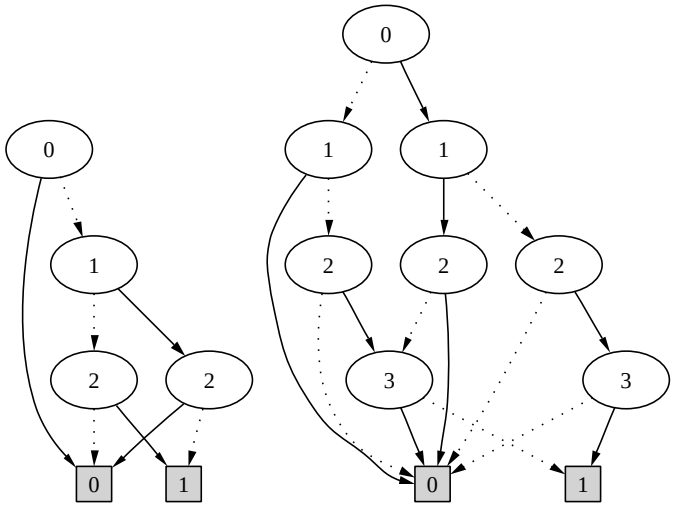

Рис. 5: Представление множеств $\{2, 3, 4, 5\}$ и $\{2, 11, 12\}$

для множества из четырех элементов, занумерованных последовательно, его размер достаточно мал. На правой части изображена BDD для подмножества занумерованного не последовательно.

4.1 Минимизация представления Kill множеств

На вход алгоритма, решающего задачу о достигающих определениях, поступает последовательность определений. Она отсортирована таким образом, что определения, осуществляющие запись в одно и то же место памяти, находятся в этой последовательности единым блоком. Во время обхода последовательности определений, можно выделить те из них, которые перетираются не достигнув выхода из линейного участка. Таким образом можно уменьшить блок определений, пишущих в одну переменную, исключив ненужные определения из рассмотрения.

Полученные блоки определений, пишущих в одну и ту же переменную и выходящих за пределы линейного участка, входят или не входят в какое-либо $KILL$ множество одновременно. Поэтому для минимизации этих множеств стоит нумеровать такие блоки последовательно.

Как показала апробация на реальных программах, использование описанной нумерации приводит к компактным представлениям $KILL$ множеств. Причем размер представлений GEN множеств, которые имеют меньшую мощность нежели $KILL$ множества, остаются довольно компактными.

Однако описанная минимизация $KILL$ и GEN множеств привела

к тому, что размер IN и OUT множеств для метода основанного на представлении множеств с помощью BDD выросли, что привело к замедлению работы метода. Так же вырос пиковый размер *GlobalSolution* отношения для метода использующего глобальные BDD.

4.2 Минимизация представления Gen множеств

Следующим, опробованным методом нумерации определений стал метод ориентированный на минимизацию GEN множеств.

Так же как и в случае с минимизацией $KILL$ множеств, мы не рассматриваем определения не достигающие выхода из своего линейного участка. Определения присваивающие какие-то значения одной и той же переменной не будут иметь последовательную нумерацию, поскольку любая вершина граф потока управления может генерировать только одну из этих переменных и последовательная нумерация не приведет к компактному представлению GEN множества какой либо вершины. Поэтому будем нумеровать определения находящиеся в каком-либо линейном участке программы последовательно.

Использование такой нумерации сократило размер представлений GEN множеств по отношению к предыдущему методу. Однако, вместе с уменьшением представлений GEN множеств, существенно выросли представления $KILL$ множеств (в несколько раз). Но несмотря на такое увеличение, пиковый размер представлений как IN и OUT множеств, так и *GlobalSolution* во время процесса итерирования сократился, что привело к уменьшению времени работы алгоритмов.

4.3 Практические результаты

Замеры производительности различных методов решения задачи о достигающих определениях осуществлялись на реальных программах, используемых для тестирования системы “Modernization Workbench”. Методы сравнивались по времени работы и максимальному количеству используемой памяти. Сравнивались не только методы, но и влияние различных нумераций на производительность.

Наилучшей из апробированных нумераций оказалась нумерация минимизирующая размеры представлений GEN множеств. Эта нумерация хорошо сказалась на производительности обоих методов, основанных на использовании BDD.

К сожалению, в рамках данной дипломной работы, автору не удалось найти способа закодировать отношения принадлежности определений к узлам графа потока управления таким образом, чтобы метод

использующий глобальные BDD оказался быстрее классического метода. Однако метод основанный на замене битовых векторов на BDD показал свою эффективность, особенно при анализе программ большого размера.

В таблице 1 приведены результаты запусков классического алгоритма анализа достигающих определений, а также алгоритма основанного на замене битовых векторов диаграммами двоичных решений и нумерацией определений, ориентированной на минимизацию GEN множеств. Графическое представление этих данных находится в приложении на рисунках 6 и 7. Полученные данные показывают, что BDD-представление

Программа	битовые вектора		BDD	
	время (с)	память (MB)	время (с)	память (MB)
LNTP	298	651	219	428
T2TP	119	508	115	339
utl2010	112	949	67	464
IQTP	107	494	105	332
cl103a	29	534	31	335
smaccum	14	234	15	230

Таблица 1: Производительность методов. “BDD” – метод основанный на итерациях по CFG с представлением множеств в виде BDD. “Битовые вектора” – классический метод. Время указано в секундах, а память в мегабайтах.

множеств хранящих информацию о потоке данных настолько компактно, что не только занимает меньше памяти по сравнению с битовыми векторами, но и позволяет эффективно оперировать ими.

5 Заключение

В рамках данной дипломной работы был проведен анализ способов применения BDD в анализе программ. Были рассмотрены два метода решения задачи о достигающих определениях, основанные на использовании диаграмм двоичных решений. Были разработаны способы минимизации BDD, используемых во время работы алгоритмов.

Рассмотренные алгоритмы были реализованы в промышленной системе реинжиниринга программ. При реализации был разработан интерфейс с библиотекой диаграмм двоичных решений. Практические результаты были получены для большого количества реальных программ. Оба предложенных метода оказались эффективнее классического метода анализа с точки зрения количества расходуемой памяти. Один из методов превзошел классический метод по времени работы. Планируется его использование в следующей версии системы реинжиниринга программ “Modernization Workbench”.

Список литературы

[1] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[2] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):958–999, 1959.

[3] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), 1978.

[4] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[5] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[6] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.

[7] Jawahar Jain, William Adams, and Masahiro Fujita. Sampling schemes for computing obdd variable orderings. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 631–638, New York, NY, USA, 1998. ACM Press.

[8] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds, 2003.

[9] J. Ullman A. Aho, R. Sethi. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[10] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[11] Devasish Das. Development of a binary decision diagram solver for live variables analysis. Tata reseach design and development centre, 2003.

[12] Relativity Modernization Workbench User Manual. Available with the Modernization Workbench system ©Relativity Technologies `http://www.relativity.com/`.

[13] Jorn Lind-Nielsen. Buddy: Binary decision diagram package. ITUniversity of Copenhagen (ITU), 2002.

[14] Graphviz. http://www.graphviz.org.

A Приложение

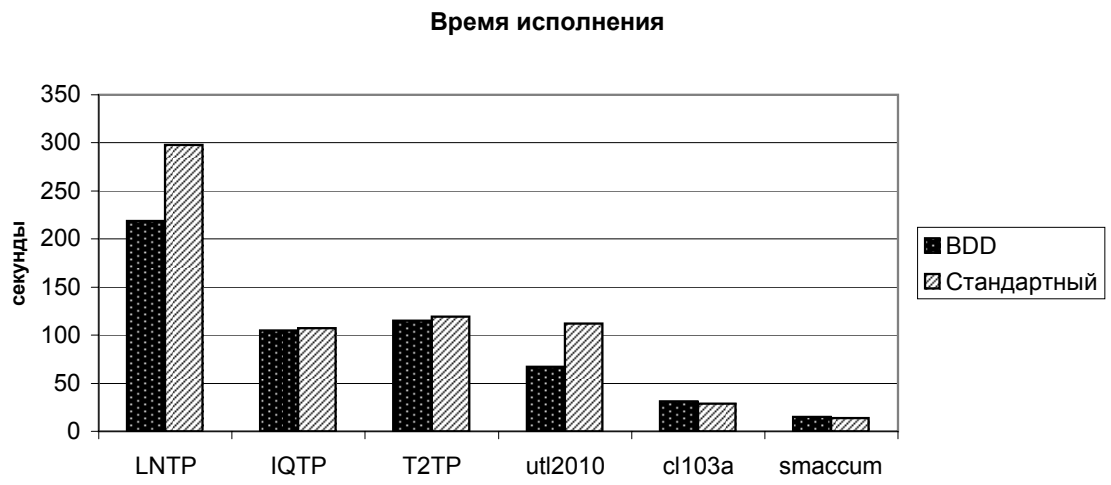

Рис. 6: Производительность методов. Скорость работы.

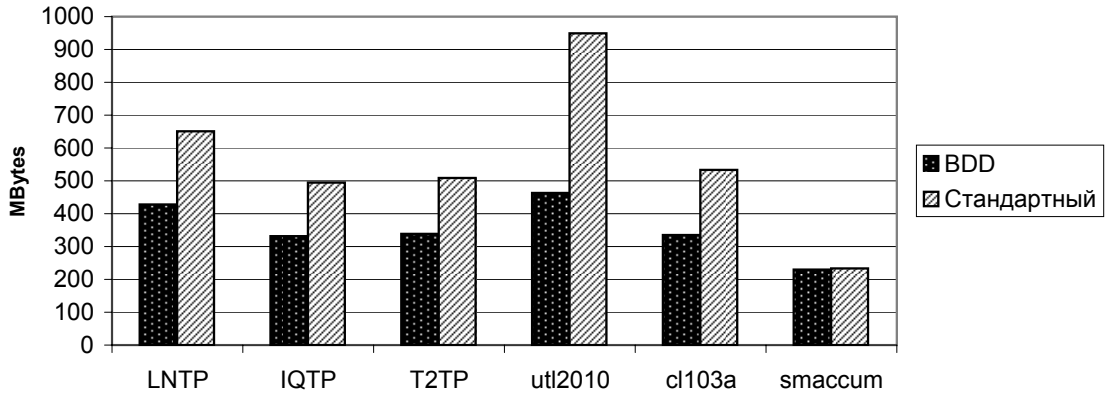

Рис. 7: Производительность методов. Затраты памяти.