Оптимизация задачи проверки выполнимости булевских ограничений при помощи кэширования промежуточных результатов

С. П. Вартанов
svartanov@ispras.ru

6 февраля 2012 г.

Аннотация

В статье предложена оптимизация алгоритма проверки выполнимости булевых формул DPLL (Davis — Putnam — Logemann — Loveland) с помощью кэширования промежуточных результатов при решении задачи нахождения входных данных для неинтерактивных программ. Дополнительная информация для оптимизации алгоритма запоминается на одном из предыдущих запусков алгоритма. Возможность подобного рода модификации алгоритма основана на особенности последовательностей проверяемых формул.

В результате модифицированный алгоритм показал ускорение по сравнению с использованием алгоритма DPLL без оптимизаций. Для проверки использовался тестовый солвер, последовательности формул генерировались инструментом Avalanche.

1 Введение

1.1 Задача проверки выполнимости булевых формул

Задача проверки выполнимости булевых формул (SAT, или задача ВЫП) формулируется следующим образом:

> Задано множество булевых переменных и определено некоторое выражение над ними. Необходимо определить, существует ли та-

кой набор значений этих переменных, при которых выражение выполнимо, т. е. принимает значение «истина».

Согласно теореме Кука, доказанной им в 1971-м году, задача SAT NP-полна. Она также остаётся NP-полной, если выражение имеет вид конъюнктивной нормальной формы от своих переменных. [4]

Тема сведения задач из различных областей прикладной математики и информатики к задаче SAT хорошо развивается. Одна из причин этого — возможность сведения к ней задач из класса NP за полиномиальное время и тем самым возможность решать достаточно трудные задачи, возникающие в самых различных областях, единым алгоритмом решения задачи SAT.

1.2 Алгоритм DPLL

Одним из наиболее эффективных алгоритмов проверки выполнимости булевых формул является алгортим DPLL (Davis — Putnam — Logemann — Loveland), основанный на бэк-трекинге. Суть алгоритма заключается в том, что для исходной булевой формулы запускается рекуррентная функция, производящая над формулой следующие действия: [3]

Распространение «единичных» конъюнктов (unit propogation). Если в КНФ присутствует конъюнкт, состоящий из одного литерала (отрицания литерала), он удаляется из КНФ, а все вхождения литерала заменяются на true (false).

Удаление «чистых» литералов (pure literal elimination). Производится поиск в формуле «чистых» литералов — переменные в которых встречаются только без отрицаний или только с ними — и вместо них подставляются значения true или false соответственно. Несмотря на то, что этот шаг является частью оригинального алгоритма, на практике, обычно, его опускают, поскольку накладные расходы зачастую превосходят эффект от его применения.

Выбор очередного литерала и запуск рекуррентной функции от формулы, в которой выбранная литера заменена на true и запуск функции, в которой она заменена на false.

На вход алгоритм принимает булеву формулу в конъюнктивной нормальной форме.

Алгоритм прекращает работу, либо когда на одной из веток значение формулы стало равным true, либо когда полный обход дерева показал, что присвоить значения переменным таким образом, чтобы значение формулы стало равным истине, невозможно.

2 Постановка задачи

Существует проект Avalanche, в котором для поиска ошибок и уязвимостей в программах происходит подбор входных данных. Для неинтерактивных программ для этого необходимо проверить возможность истинности всех условных переходов, встретившихся в трассе программы. Построенные формулы могут быть конвертированы в конъюнктивные нормальные формы. В этом случае задача сводится к проверке выполнимости их конъюнкции, т. е. к проблеме SAT.

Последовательность ограничений, подаваемых на вход солверу STP, а также генерируемая им последовательность булевых формул, имеет некоторые особенности. В частности, две последовательно идущие формулы зачастую имеют одинаковые части.

Необходимо разработать алгоритм проверки выполнимости булевых формул, использующий информацию, полученную при проверке выполнимости одной формулы из последовательности для ускорения проверки выполнимости другой формулы.

2.1 Требования к решению

Алгоритм должен быть основан на использующимся в решателе MiniSat алгоритме DPLL. Также он должен опираться на особенности последовательностей формул, для которых решается задача проверки выполнимости.

3 Исследование и построение решения задачи

3.1 Проект Avalanche

Avalanche — инструмент обнаружения программных дефектов при помощи динамического анализа. Инструмент использует возможности динамической инструментации программы, представляемые Valgrind, для сбора и анализа трассы выполнения программы. [1]

На одном из этапов работы программы, инструменту необходимо проверить набор булевых ограничений.

Для решения задачи выполнимости булевых ограничений в проекте используется солвер STP. Схема работы солвера представлена ниже. [6]

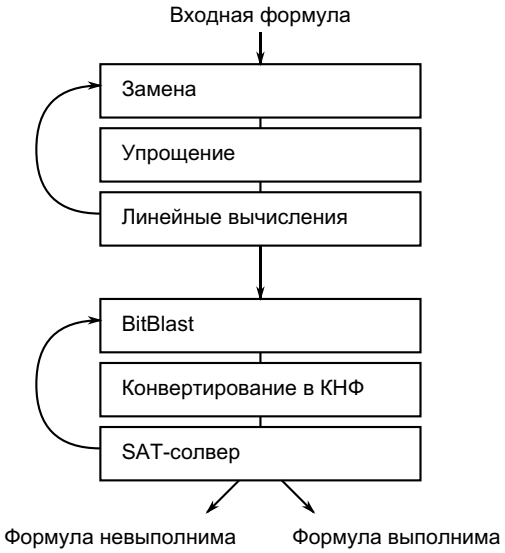

Рис. 1: Работа солвера STP

Решение этой задачи занимает значительную часть работы. Так, на половине проектов, которые были проанализированы инструментом Avalahche, работа STP заняла больше 70 % всего времени, а на таких проектах как «sndfile-mix-to mono» и «cjpeg», солвер работал более 99 % от времени работы инструмента[1]. Следовательно, существует потребность в оптимизации алгоритма решения этой задачи.

STP выполняет некие преобразованя входных формул, в том числе преобразование выражений в КНФ, кототрые он подаёт на вход решателю MiniSat.

3.2 Модификация алгоритма DPLL

В решателе MiniSat используется рассмотренный ранее алгоритм DPLL. [5]

Поскольку задача SAT является в общем случае труднорешаемой, стоит попробовать оптимизировать алгоритм с учётом особенностей формул,

выполнимость которых необходимо проверять. Например, для автоматического определения выполнимости наборов формул для операций сравнения используются особенности этих операций. [2]

Зачастую требуется проверить набор ограничений следующего вида:

$$\begin{cases} \overline{A_1(x_1)}; \\ A_1(x_1) \wedge \overline{A_2(x_2)}; \\ \dots \\ A_1(x_1) \wedge A_2(x_2) \wedge \dots \wedge \overline{A_n(x_n)}. \end{cases}$$

Эта особенность формул основана на виде трасс, для которых проверяются ограничения. Рассмотрим простой пример.

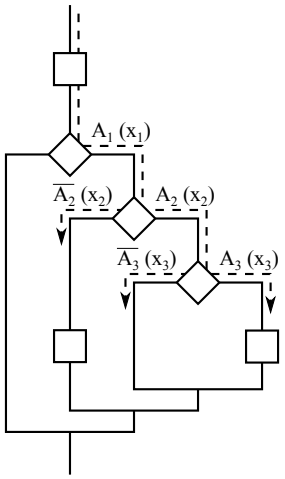

Рис. 2: Пример графа потока управления

На схеме представлен граф потока управления некоторой программы. В ходе динамического анализа существует необходимость покрыть как можно больше операторов программы. Для этого необходимо выяснить, какие ограничения (например, значения переменных) должны выполняться, чтобы выполнения программы шло по одному из возможных путей.

На представленном примере возможны три выполнения программы. Им соответствуют следующие ограничения:

$$\begin{cases} \overline{A_1(x_1)}; \\ A_1(x_1) \wedge \overline{A_2(x_2)}; \\ A_1(x_1) \wedge A_2(x_2) \wedge \overline{A_3(x_3)}. \end{cases}$$

Продолжим рассмотрение общей формулы, и поскольку выполнимость формулы и выполнимость её отрицания — вещи, связанные слабо, преобразуем систему к более общему виду:

$$\begin{cases} B_1; \\ A_1 \wedge B_2; \\ \dots \\ A_1 \wedge A_2 \wedge \dots \wedge B_n. \end{cases}$$

Заметим, что формулы $m-1$ и m, соответственно, имеют вид:

$$A_1 \wedge A_2 \wedge \dots \wedge A_m \wedge B_{m+1}$$
$$\underbrace{A_1 \wedge A_2 \wedge \dots \wedge A_m}_{equals} \wedge A_{m+1} \wedge B_{m+2}$$

При достаточно больших m одинаковая часть обеих формул значительно превышает части, в которых они различаются.

Попробуем использовать информацию, полученную при проверке выполнимости формулы A для проверки выполнимости $B = A \wedge C$.

Преобразуем алгоритм следующим образом. Во время проверки выполнимости формулы A, запомним место в дереве, на котором мы остановились и состояние в нём, а также все состояния всех точек в которых остались непросмотренные ветки и в которые мы можем вернуться в случае неудачи на текущей ветке.

3.3 Пример работы алгоритмов

Рассмотрим предложенный алгоритм на двух следующих конкретных формулах:

$$A = (x_0 \vee x_1) \wedge (x_0 \vee \overline{x_1}) \wedge (\overline{x_0} \vee x_2) \wedge (\overline{x_3} \vee x_4),$$
$$B = (x_0 \vee x_1) \wedge (x_0 \vee \overline{x_1}) \wedge (\overline{x_0} \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (x_3 \vee x_5) \wedge (\overline{x_5}).$$

Формула B есть конъюнкция формулы A и $C = (x_3 \vee x_5) \wedge (\overline{x_5})$. Рассмотрим, сначала, как работает алгоритм DPLL без pure literal assign для формулы A.

Сплошными линиями на схеме показана работа алгоритма для формулы A. В вершине 3 формула принимает вид $\text{false} \wedge (\overline{x_3} \vee x_4)$ и тождественно равна false. После этого вычисление возвращается к последней нерассмотренной ветке, а именно, выбору true в качестве значения переменной x_0. В вершине 6 формула тождественно равна true, следовательно, исходная формула выполнима (значения переменных можно проследить, пройдя из этой вершины к корню). При этом осталась нерассмотренной ветка, выходящая из вершины 5, в которой переменной x_3 присваивается значение true.

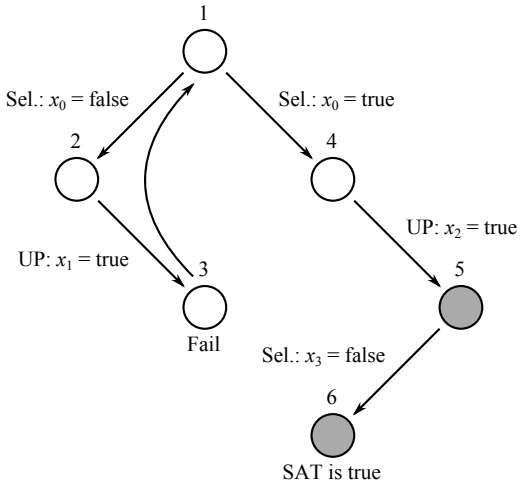

Рис. 3: Пример работы алгоритма DPLL. Метками Sel. и UP обозначены присваивания, происходящие на шагах выбора переменной и распространения «единичных» конъюнктов, соответственно.

Рассмотрим теперь работу алгоритма при проверке выполнимости формулы B. Вычисление начнём не с корня дерева, а с точки, на которой мы закончили вычисление при проверке выполнимости формулы A. При этом мы сохранили значения формулы и переменных в вершине 6, а также в вершине 5 — единственной, оставшейся непросмотренной. На схеме вершины, в которых запомнены состояния закрашены.

Например, состояния в вершинах 5 и 6 следующие:

Дальнейшее вычисление на схеме показано сплошными линиями. Добавим к формуле в вершине 6 конъюнкцию $(x_3 \vee x_5) \wedge (\overline{x_5})$ и присвоим переменным значения из состояния в этой вершине. Получим формулу $(x_5) \wedge (\overline{x_5})$. Последующее продвижение единичных конъюнктов в вершине 7 показыва-

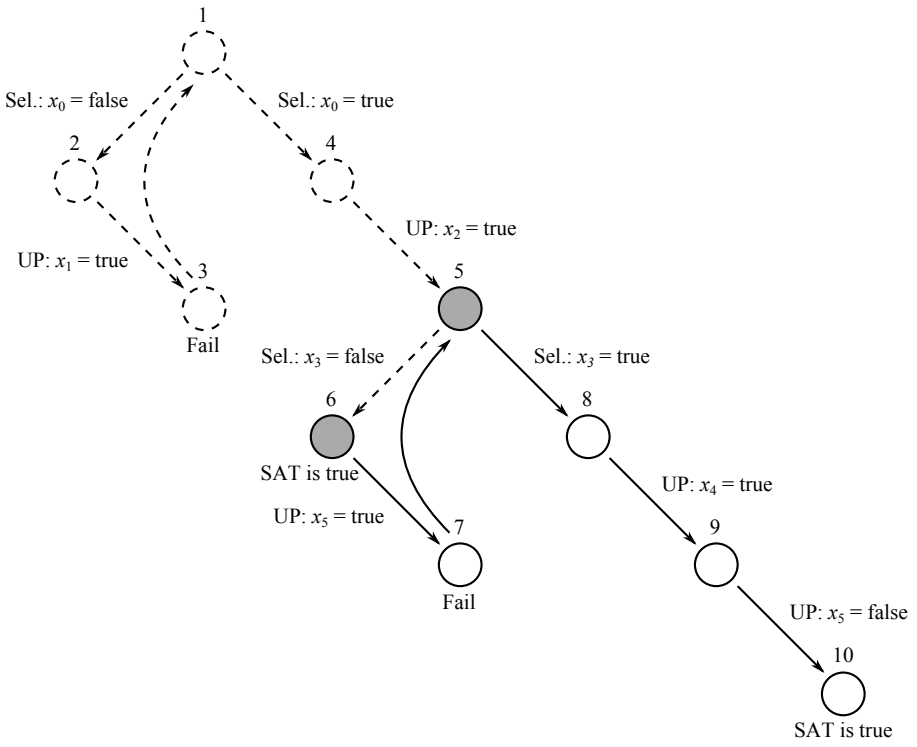

Рис. 4: Пример работы модифицированного алгоритма DPLL

ет, что формула равна false. Выполнимость формулы B в этой вершине не доказана. Вернёмся к последней нерассмотренной вершине (вершина 5) и продолжим вычисление с этой точки. В вершине 10 формула тождественно равна true и выполнимость формулы B доказана.

Таким образом, для формул

$$B_0 \wedge B_1 \wedge \cdots \wedge B_{m-1} \wedge A_m$$
$$B_0 \wedge B_1 \wedge \cdots \wedge B_{m-1} \wedge B_m \wedge A_{m-1}$$

можно сначала проверить выполнимость формулы $B_0 \wedge B_1 \wedge \cdots \wedge B_{m-1}$, т. е. общей части, а затем вышеуказанным методом проверить выполнимость формул вцелом.

Поскольку алгоритм выполняет некоторые дополнительные действия, необходимо проверить, оправданы ли они.

Состояние	x_0	x_1	x_2	x_3	x_4	x_5	Формула
5	true	—	true	—	—	—	$(x_2) \wedge (\overline{x_3} \vee x_4)$
6	true	—	true	false	—	—	true

Таблица 1: Состояния

4 Тестовый солвер

Для проверки алгоритма на практике, был написан тестовый солвер на языке Java. Он реализует алгоритм DPLL без шага, на котором удаляются «свободные» литералы. Выбор очередного литерала выполняется только после того, как шаг продвижения единичных конъюнктов более неприменим. Такой подход используется в большинстве реализация алгоритма DPLL.

В солвере используется единственная оптимизация алгоритма DPLL — в памяти для каждой переменной хранится число вхождений этой переменной в формулу с отрицаниями и без.

Также солвер реализует описанную модификацию алгоритма. После каждой проверки формулы, если выполнимость её подтверждена, ветка дерева разбора на которой было найдено решение, а также все наборы присваиваний в вершинах, ответвления из которых остались непросмотренными, сохраняются в памяти.

Программа работает с формулами в конъюнктивной нормальной форме, записанными в текстовый файл. Переменные обозначаются символом «`x`» с индексом, отрицание знаком «`!`», логическое сложение — знаком «`|`», логическое умножение — знаком «`&`». Поскольку формула задаётся только в КНФ, скобки опускаются.

Например, формула

$$(x_0 \vee x_1) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_3 \vee x_4)$$

должна быть записана следующим образом: `x0|x1&!x1|x2|!x3&x3|x4`.

Пример вывода солвера:

```
(x0 | x1) & (x0 | !x1) & (!x0 | x2) & (!x3 | x4)
After UP - (x0 | x1) & (x0 | !x1) & (!x0 | x2) & (!x3 | x4)
Select 0
Assign (Select) x0 = false
(x1) & (!x1) & (!x3 | x4)
```

```
Assign (UP) x1 = true
After UP - () & (-x3 | x4)
SAT is false.

Assign (Select) x0 = true
(x2) & (!x3 | x4)
Assign (UP) x2 = true
After UP - (!x3 | x4)
Select 3
Assign (Select) x3 = false
SAT is true: 1210 - 0, -> 2, => 3, <-
```

В последней строке содержится информация о том, что для формулы установлена её выполнимость. Там же содержится вектор присваиваний - `1210`, где каждая цифра соответствует значению переменной (true для переменных с номерами 0 и 2, false для переменной с номером 3 и переменная 1 осталась без присваивания). Строка `0, -> 2, => 3, <-` — описание ветки, на которой было найдено решение. Все эти результаты сохраняются в памяти.

5 Проверка работы модифицированного алгоритма

5.1 Трассы инструмента Avalanche

Скорость работы предложенной модификации алгоритма была проверена на последовательностях трасс, передаваемых инструментоми Avalanche солверу STP при проверке им различных программ.

Для каждого теста в ходе работы инструмента извлекались две последовательно вычисляющиеся трассы. Сначала выполнимость обеих формул независимо проверялась алгоритмом DPLL. А затем выполнялась проверка первой формулы, в ходе которой запоминались необходимые состояния и на основе этой информации запускался модифицированный алгоритм на второй формуле.

При этом последовательности утверждений были конвертированы в конъюнктивные нормальные формы.

Для каждого алгоритма указано время его выполнения на тестовом солвере в секундах и количество выполненных присваиваний (в ходе шага выбора переменной или unit propogation) во второй формуле. Число конъюнктов также указано для второй формулы.

Тест	Число конъюн.	DPLL			Модиф. DPLL			Ускорение (раз)	Итоговое ускорение
		1 ф.	2 ф.	пр.	1 ф.	2 ф.	пр.		
1	416	2.87	1.07	641	2.94	0.37	175	2.86	1.19
2	416	1.27	1.39	806	1.33	0.34	260	4	1.58
3	416	1.35	1.80	837	1.43	0.37	292	4.7	1.73
4	480	1.59	1.78	837	1.67	0.19	192	8.9	1.8
5	544	3.01	1.97	837	3.02	0.22	96	8.7	1.53
6	540	14.82	0.29	251	14.89	0.20	1	1.4	1.0007
7	362	8.15	6.8	407	8.16	0.74	71	9.17	1.67
8	1340	77.03	0.721	634	77.06	0.15	1	4.65	1.007
9	665	9.55	0.29	337	9.59	0.14	1	2	1.01
10	527	25.96	22.65	574	26.02	2.09	32	10.81	1.72
11	559	23.37	22.44	606	23.46	2.46	32	9.08	1.76
12	591	26.28	21.45	638	26.28	1.79	32	11.93	1.69

Таблица 2: Сравнение скорости работы алгоритмов.

5.2 Случайные КНФ

Также скорость работы алгоритма была проверена на случайным образом сгенерированных конъюнктивных нормальных формах.

В таблице приведены средние результаты запусков следующих формул:

1. 50 конъюнктов с не более чем 50 литералами,
2. 100 конъюнктов с не более чем 50 литералами,
3. 50 конъюнктов с не более чем 100 литералами.

Следует отметить, что в общем случае может быть выполнен полный обход ветки, которую использует алгоритм, прежде чем будет обнаружена переменная, предположение о значении которой становится неверным в «новой» формуле.

Тест	Число конъюн.	DPLL			Модиф. DPLL			Ускорение (раз)	Итоговое ускорение
		1 ф.	2 ф.	пр.	1 ф.	2 ф.	пр.		
1	50	0.05	0.16	165	0.1	0.02	73	5.85	1.61
2	100	0.2	0.31	360	0.24	0.07	189	4.48	1.64
3	50	0.15	0.30	173	0.23	0.06	95	4.98	1.56

Таблица 3: Сравнение скорости работы алгоритмов.

Однако, как видно из результатов тестов, на практике такой случай встречается редко.

Чтобы избежать падения эффективности в этих случаях, можно ограничивать глубину обхода ветки. При этом, если ограничение будет достигнуто, продолжать обход с корня, что эквивалентно запуску немодифицированного DPLL.

6 Заключение

Последовательности были проверены на тестовом солвере сначала алгоритмом DPLL, а затем его модифицированным вариантом.

После сравнения скорости работы алгоритмов, можно сделать вывод о том, что в среднем модифицированный алгоритм даёт ускорение.

Поскольку в среднем алгоритм даёт выигрыш во времени, он может быть применён для указанных выше целей.

В дальнейшие планы входит внедрение алгоритма в решатель MiniSat.

Список литературы

[1] Исаев, И. К. Сидоров Д. В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование [№ 4]. 2010. 16 с.

[2] С. В. Зеленов, С. А. Зеленова. Автоматическое определение выполнимости наборов формул для операций сравнения. Труды ИСП РАН, том 14. 109–118 с. Москва, 2008.

[3] Davis M., Logemann G., Loveland D. A machine program for theorem proving // Communication of the ACM. 1962. P. 394–397.

[4] Новикова Н. М. Основы оптимизации. Москва. 1998. 17–22 с.

[5] Eén N., Sörensson N. MiniSat solver [HTML] (http://minisat.se/)

[6] Katelman M., Soos M. STP Constraint Solver [HTML] (http://sites.google.com/site/stpfastprover/)