Записки научных
семинаров ПОМИ
Том 316, 2004 г.

**А. С. Куликов, С. С. Федин**

# АВТОМАТИЧЕСКИЕ ДОКАЗАТЕЛЬСТВА ВЕРХНИХ ОЦЕНОК НА ВРЕМЯ РАБОТЫ АЛГОРИТМОВ РАСЩЕПЛЕНИЯ

## 1. Введение

Существует несколько подходов к решению NP-трудных задач. К примеру, для некоторых задач известны алгоритмы, находящие приближенное решение, а также алгоритмы, находящие точные решения с некоторой вероятностью. Однако, на практике многие проблемы должны быть решены точно, поэтому в данной статье мы рассматриваем наиболее популярный тип алгоритмов, так называемые *алгоритмы расщепления*. Хорошо известный пример алгоритма расщепления для задачи SAT–DLL-подобный алгоритм [5].

Многие NP-трудные задачи могут быть сформулированы в терминах булевых формул (например, в статье [8] показано, как сформулировать в терминах булевых формул задачу о максимальном сечении графа). Поэтому мы рассматриваем алгоритмы расщепления для задач, имеющих дело с формулами в КНФ (например, SAT, MAXSAT). Ниже мы кратко описываем основную идею алгоритма расщепления для таких задач. На каждом шаге алгоритм сначала упрощает формулу, используя правила упрощения. Далее, если полученная формула $F$ не является тривиальной, он делает два рекурсивных вызова на формулах, полученных из $F$ присваиваниями $l = \texttt{true}$ и $l = \texttt{false}$, где $l$ – литерал формулы $F$ (как правило, такие формулы обозначаются через $F[l]$ и $F[\bar{l}]$). И наконец, алгоритм возвращает ответ, исходя из ответов, возвращенных рекурсивными вызовами. К примеру, в случае задачи SAT он возвращает ответ "Выполнима", если и только если хотя бы один из рекурсивных вызовов вернул такой

При частичной поддержке гранта No. 1 6-го конкурса-экспертизы научных проектов молодых ученых РАН (1999), а также гранта научных школ (код 3180Ш1).

ответ. В общем случае алгоритм расщепления может рекурсивно вызвать себя на большем количестве формул. Единственным условием на эти формулы является возможность построения за полиномиальное время решения для исходной формулы по решениям для этих формул.

Существует большое количество литературы, описывающей алгоритмы расщепления. Более того, для многих проблем есть целая "иерархия" статей, содержащих такие алгоритмы, где каждая статья улучшает результат предыдущей более тщательным разбором случаев или введением нового правила упрощения. К примеру, существуют алгоритмы расщепления для задач SAT (см. обзорную статью [4]), MAXSAT ([2, 3, 8]), XSAT ([1, 10]), MUX-CUT ([6, 8]), а также многих других. Как правило, размер анализа случаев в такой иерархии растет с улучшением результата. Иногда, однако, новое правило упрощения позволяет доказать лучший результат, рассмотрев меньшее количество случаев.

В данной работе мы представляем алгоритм, который по множеству правил упрощения автоматически генерирует доказательство верхней оценки на время работы алгоритма расщепления, использующего эти правила. Мы представляем несколько автоматически порожденных доказательств верхних оценок для задач SAT, MAXSAT и $(n,3)$-MAXSAT.

**Организация статьи.**

Наша статья организована следующим образом: в разделе 2 мы даем основные определения; раздел 3 содержит детальное описание нашей программы; в разделе 4 мы обсуждаем подобные работы; раздел 5 содержит дальнейшие направления (в частности, мы указываем несколько "узких" мест всех известных на данный момент программ для автоматического доказательства верхних оценок для NP-трудных задач).

## 2. Основные определения

Пусть $V$ – множество булевых переменных. Отрицание переменной $v$ обозначается через $\overline{v}$, посредством $\overline{V}$ мы обозначаем множество $\{\overline{v} \mid v \in V\}$. *Литералами* мы называем элементы множества $W = V \cup \overline{V}$. Если $w$ обозначает литерал $\overline{l}$, то $\overline{w}$ обозначает литерал $l$. *Клоз* есть дизъюнкция конечного числа литералов, не содержащая ни одной переменной одновременно с ее отрицанием.

Пустой клоз интерпретируется как `false`. *Формула в КНФ* – это конъюнкция конечного числа клозов. Пустая формула интерпретируется как `true`. *Длина клоза* есть количество его литералов, *длина формулы* – сумма длин ее клозов. Мы говорим, что *литерал $l$ встречается* в клозе или формуле, если данный клоз или данная формула содержит литерал $l$. Однако, когда мы говорим, что *переменная $v$ встречается* в клозе или формуле, мы имеем в виду, что данный клоз или данная формула содержит литерал $v$ или литерал $\overline{v}$. $(i, j)$*-литерал* есть литерал, встречающийся в формуле $i$ раз положительно и $j$ раз отрицательно.

*Набором* мы называем конечное подмножество множества $W$, не содержащее ни одной переменной одновременно с ее отрицанием. Говоря неформально, то, что набор $I$ содержит литерал $l$, означает, что литералу $l$ присвоено значение `true` в $I$. Для получения формулы $F[I]$ из формулы $F$ и набора $I = \{l_1, \ldots, l_s\}$, мы сначала удаляем из $F$ все клозы, содержащие литералы $l_i$, а потом удаляем все вхождения литералов $\overline{l_i}$ из оставшихся клозов.

**2.1. Оценка размера дерева расщепления.**

Кульман и Люкхардт в работе [11] ввели понятие дерева расщепления. Работу алгоритма расщепления можно рассматривать как дерево, листья которого помечены формулами в КНФ, так что если вершина помечена формулой $F$, то ее сыновья помечены упрощенными формулами $F[I_1], F[I_2], \ldots, F[I_k]$ для некоторых наборов $I_1, I_2, \ldots, I_k$. Каждой формуле такого дерева мы приписываем неотрицательное целое число $\mu(F)$, обозначающее *сложность* $F$. В нашей программе мы используем следующие меры сложности:

1. $\mu(F) = N(F)$ – количество переменных $F$;
2. $\mu(F) = K(F)$ – количество клозов $F$;
3. $\mu(F) = L(F)$ – длина $F$.

Дерево является *деревом расщепления*, если сложность формулы каждой вершины строго больше сложности каждой из формул, которыми помечены сыновья этой вершины.

Рассмотрим вершину нашего дерева, помеченную формулой $F_0$. Пусть ее сыновья помечены формулами $F_1, F_2, \ldots, F_m$. *Вектором расщепления* этой вершины называется вектор $(t_1, t_2, \ldots, t_m)$, в котором $t_i$ есть натуральное число, не превосходящее $\mu(F_0) - \mu(F_i)$. Характеристический многочлен данного вектора расщепления определяется следующим образом: $h(x) = 1 - \sum_{i=1}^{m} x^{-t_i}$. Единственный положительный корень этого многочлена называется *числом расщепления* и обозначается через $\tau(t_1, t_2, \ldots, t_m)$. *Числом расщепления дерева* $T$ называется максимальное из чисел расщепления всех его вершин; мы обозначаем его через $\tau_{\max,T}$.

На данный момент мы дали все необходимые определения для предоставления леммы, доказанной Кульманом и Люкхардтом [11], которая позволяет оценить количество листьев дерева расщепления, используя его число расщепления.

**Лемма 2.1.** *Пусть $T$ – дерево расщепления, корень которого помечен формулой $F_0$. Тогда его количество листьев не превосходит* $(\tau_{\max,T})^{\mu(F_0)}$.

Ниже мы показываем, каким образом эта лемма помогает оценить время работы алгоритма расщепления. Как уже было упомянуто, алгоритм расщепления сначала упрощает входную формулу, после чего производит несколько рекурсивных вызовов на формулах меньшей сложности. Понятно, что общее время работы такого алгоритма складывается из времени работы всех рекурсивных вызовов, а также времени, потраченного на то, чтобы произвести эти вызовы (отметим, что мы рассматриваем только алгоритмы, которые на каждом шаге производят рекурсивные вызовы, количество которых ограничено некоторой константой). Таким образом, с точностью до многочлена время работы алгоритма есть количество вершин (или листьев) дерева рекурсии.

### 2.2. Задачи SAT, MAXSAT и $(n, 3)$-MAXSAT.

В данном подразделе мы даем определения NP-трудных задач, на которых мы тестировали нашу программу.

Задача пропозициональной выполнимости (SAT) заключается в проверке существования для данной формулы в КНФ набора булевых значений для всех ее переменных, который выполняет все клозы данной формулы. В задаче максимальной выполнимости (MAXSAT) требуется найти максимальное количество клозов данной формулы, которое может быть одновременно (то есть одним набором) выполнено. Задача $(n, 3)$-MAXSAT есть частный случай задачи MAXSAT, где каждая переменная появляется во входной формуле не более, чем три раза.

Лучшие на данный момент верхние оценки для данных проблем приведены в таблице 1. Все эти оценки получены при помощи метода расщепления. Отметим, что пока не известно оценок в форме $O(c^N)$, где $c < 2$ есть константа, для задач SAT и MAXSAT. Мы не знаем оценок для $(n,3)$-MAXSAT относительно количества клозов и длины формулы, улучшающих соответствующие оценки для MAXSAT.

Таблица 1. Лучшие известные оценки для SAT, MAXSAT и $(n,3)$-MAXSAT.

| | $N$ | $K$ | $L$ |
|---|---|---|---|
| SAT | $O(2.000000^N)$ | $O(1.238823^K)$ [9] | $O(1.073997^L)$ [9] |
| MAXSAT | $O(2.000000^N)$ | $O(1.341294^K)$ [3] | $O(1.105729^L)$ [2] |
| $(n,3)$ – MAXSAT | $O(1.324719^N)$ [2] | $O(1.341294^K)$ [3] | $O(1.105729^L)$ [2] |

## 3. Алгоритм

Сначала мы даем несколько нестандартных определений, которые удобны в контексте данной работы, а потом представляем основной алгоритм нашей программы.

Типичный алгоритм расщепления сначала упрощает входную формулу, после чего расщепляет полученную формулу. Как правило, он расщепляет в зависимости от некоторых свойств формулы. То есть в описании алгоритма расщепления применение правил упрощения предшествует нескольким случаям следующей формы: “если формула содержит такие клозы или литералы: ..., то расщепить следующим образом: ...”. Для доказательства корректности такого алгоритма необходимо доказать, что каждая упрощенная формула (мы называем формулу упрощенной, если ни одно правило упрощения к ней не применимо) удовлетворяет условию хотя бы одного из этих случаев. Отметим, что каждому такому случаю соответствует множество формул, удовлетворяющих условию этого случая, и алгоритм корректен, если объединение всех таких множеств содержит множество упрощенных формул. В следующем подразделе мы вводим удобное для описания алгоритмов расщепления понятие класса формул.

### 3.1. Класс формул.

Рассмотрим клоз $C$, состоящий из литералов $l_1, l_2, \dots, l_k$. *Клозом с неопределенной длиной* мы называем множество клозов, каждый из которых содержит все перечисленные литералы, а также, возможно, еще какие-нибудь литералы. Обозначается такое множество следующим образом: $(l_1 \vee l_2 \vee \ldots \vee l_k \ldots)$. Литералы $l_1, l_2, \ldots, l_k$ мы называем *базисом* этого клоза. К примеру, $(l \ldots)$ есть множество всех клозов, содержащих литерал $l$, базисом этого клоза является $\{l\}$. В дальнейшем мы используем слово "клоз" для обозначения как клозов в стандартном определении, так и клозов с неопределенной длинной.

Аналогично мы определяем *класс формул*. Рассмотрим клозы $C'_1, \ldots, C'_k$ (некоторые из них могут быть с неопределенной длиной). Тогда *класс формул относительно* $C'_1, \ldots, C'_k$ есть множество формул, каждая из которых представляется, как $C_1 \wedge \ldots \wedge C_m$, и при этом выполняются следующие условия:

1. $m \geq k$,
2. для всех $i$, таких что $1 \leq i \leq k$, $C'_i \subseteq C_i$ (как множества литералов), если $C'_i$ является клозом с неопределенной длиной, и $C_i = C'_i$ в противном случае,
3. для всех $i, j$, таких что $1 \leq i \leq k$, $k + 1 \leq j \leq m$, $C_j$ не содержит ни одной переменной базиса $C'_i$.

Мы обозначаем это множество через $C'_1 \wedge \ldots \wedge C'_k \ldots$ и называем клозы $C'_1, \ldots, C'_k$ *базисом* данного класса. Мы говорим, что переменные клозов $C'_1, \ldots, C'_k$ являются *известными* для данного класса.

Говоря неформально, можно получить класс формул, заменив все вхождения некоторого множества переменных формулы знаком "...". К примеру, формула $(x \vee y \vee z) \wedge (\overline{x}) \wedge (\overline{y} \vee z \vee u) \wedge (\overline{u} \vee x) \wedge (\overline{z})$ является элементом следующего класса формул: $(z \ldots) \wedge (z \vee u \ldots) \wedge (\overline{u} \ldots) \wedge (\overline{z}) \ldots$.

Заметим, что в большинстве ситуаций мы можем работать с классом формул так же, как и с формулой в КНФ. К примеру, если мы удалим из базиса класса формул все клозы, содержащие литерал $x$, после чего удалим все вхождения литерала $\overline{x}$ из оставшихся клозов, то мы получим класс формул, получающийся из исходного класса присваиванием значения `true` литералу $x$. Легко видеть также, что если после присваивания булевого значения литералу класса формул или применения (рассматриваемого в данной статье) правила упрощения к нему сложность класса уменьшилась на $\Delta$, то сложность каждой формулы этого класса уменьшилась *хотя бы* на $\Delta$.

Для класса формул $\mathcal{F}$ и клоза $C$ мы определяем класс формул $\mathcal{F} + \{C\}$ как класс, получающийся из $\mathcal{F}$ прибавлением клоза $C$ к его базису. Аналогично определяется клоз с неопределенной длиной $C + \{l\}$ для клоза $C$ с неопределенной длиной и литерала $l$. Под *фиксированием длины клоза* $(l_1 \vee \ldots \vee l_k \ldots)$ мы понимаем замену этого клоза на клоз $(l_1 \vee l_2 \vee \ldots \vee l_k)$.

Мы говорим, что правило упрощения применимо к классу формул, если это правило применимо к каждой формуле этого класса. К примеру, каждая формула класса $(x) \wedge (x \vee \overline{y} \ldots) \wedge (y \ldots) \ldots$ содержит чистый литерал (то есть литерал, отрицание которого не встречается в формуле). Аналогично, мы говорим, что для класса формул существует расщепление, если такое расщепление существует для каждой формулы этого класса. Например, для каждой формулы класса $(x \vee y) \wedge (\overline{x} \vee y \vee \overline{t} \ldots) \wedge (\overline{y} \vee u) \wedge (u \vee \overline{z}) \wedge (\overline{x} \vee t) \wedge (\overline{u} \vee \overline{y} \vee z) \ldots$ существуют $(1, 2)$-расщепление (по переменной $x$) относительно числа переменных (при присваивании значения `false` литералу $x$ удаляется переменная $t$), $(2, 3)$-расщепление (по $y$) относительно числа клозов и $(4, 3)$-расщепление (по $z$) относительно длины формулы. Однако, при работе с конкретной формулой, как правило, возможно найти лучшие расщепления (то есть расщепления с меньшим числом расщепления), поскольку правила упрощения могут уменьшить сложность формулы после расщепления.

**3.2. Правила упрощения.**

Мы говорим, что правило упрощения применимо к формуле $F$, если оно может за полиномиальное время заменить $F$ формулой $F'$, так чтобы выполнялись два следующих условия:

- сложность $F'$ меньше сложности $F$,
- решение для $F$ может быть за полиномиальное время построено из решения для $F'$.

Мы говорим, что правило упрощения применимо к классу формул, если оно применимо к каждой формуле этого класса.

В нашей программе мы используем следующие правила упрощения ($\mathcal{F}$ обозначает класс формул):

1. *Чистый литерал.* Если $\mathcal{F}$ содержит $(k, 0)$-литерал $l$, заменить $\mathcal{F}$ на $\mathcal{F}[l]$. Действительно для всех рассматриваемых задач.

2. *Единичный клоз.* Если $(l) \in \mathcal{F}$, заменить $\mathcal{F}$ на $\mathcal{F}[l]$. Действительно только для SAT.

3. *Резолюция.* Если $\mathcal{F}$ содержит $(1,1)$-литерал $l$ и $l \in C_1$ и $\bar{l} \in C_2$, заменить эти клозы клозом $C_1 \cup C_2 - \{l, \bar{l}\}$ (если этот клоз пуст или содержит пару дополняющих друг друга литералов, то исходные клозы просто удаляются). Действительно для всех рассматриваемых задач. (На самом деле, здесь мы приводим упрощенную версию правила, более же детально оно описано, например, в [9].)

4. *Доминирующий единичный клоз.* Если $\mathcal{F}$ содержит литерал $l$, такой что литерал $\bar{l}$ встречается в $\mathcal{F}$ не больше, чем литерал $l$ встречается в единичных клозах, заменить $\mathcal{F}$ на $\mathcal{F}[l]$. Действительно для MAXSAT и $(n,3)$-MAXSAT.

5. *Замкнутая подформула.* Если $\mathcal{F}$ содержит замкнутую подформулу $F'$, заменить $\mathcal{F}$ на $\mathcal{F} - F'$. Действительно для всех рассматриваемых задач.

6. *Почти доминирующий единичный клоз.* Если $\mathcal{F}$ содержит литерал $l$, такой что $l$ встречается в $k$ единичных клозах $\mathcal{F}$, а $\bar{l}$ встречается в $k+1$ клозах, два из которых содержат пару дополняющих друг друга литералов (что означает, что один из этих клозов выполнен *всегда*), заменить $\mathcal{F}$ на $\mathcal{F}[l]$. Действительно для MAXSAT и $(n,3)$-MAXSAT.

7. *Почти общие клозы.* Если $\mathcal{F}$ содержит два клоза $C_1$, $C_2$ с определенной длиной, таких что для некоторого литерала $l$ выполнено $C_1 - \{l\} = C_2 - \{\bar{l}\}$, заменить их на клоз $C_1 - \{l\}$. Действительно для MAXSAT и $(n,3)$-MAXSAT.

8. *Выполняющий набор.* Если существует набор для всех известных переменных $\mathcal{F}$, выполняющий все клозы базиса $\mathcal{F}$, подставить этот набор в $\mathcal{F}$ (то есть присвоить значение `true` всем литералам этого набора). Действительно для всех рассматриваемых задач.

Все правила, кроме последнего, могут быть найдены в [2, 8, 9], корректность последнего правила очевидна.

**3.3. Реализация и структуры данных.**

В этом подразделе мы описываем реализацию основных процедур и структуры данных нашей программы.

Клоз представляется списком литералов и флагом `IsLength-Fixed`. Когда этот флаг установлен, литералы не могут быть добавлены в клоз. По умолчанию флаг не установлен и клоз рассматривается как клоз с неопределенной длиной. Основными операциями клоза являются фиксирование его длины, а также добавление и удаление литерала.

Правило упрощения реализовано как процедура, принимающая на вход класс формул. Она проверяет, выполняется ли ее собственное условие для входного класса, и, если выполняется, производит соответствующее упрощение формулы. Например, правило *чистый литерал* проверяет, содержит ли класс $(k, 0)$-литералы и присваивает значение `true` всем таким литералам.

Класс формул представляется как список клозов. Основной операцией класса формул является присваивание булевого значения ее литералу. При присваивании значения `true` литералу $l$ мы сначала удаляем из класса формул все клозы, содержащие литерал $l$, после чего удаляем все вхождения литерала $\overline{l}$.

Другой важной операцией класса формул является нахождение его возможных расщеплений. Наша программа рассматривает следующие расщепления ($\mathcal{F}$ – класс формул, $x, y$ – его литералы):

1. $\mathcal{F}[x], \mathcal{F}[\overline{x}]$;
2. $\mathcal{F}[x, y]$, $\mathcal{F}[x, \overline{y}]$, $\mathcal{F}[\overline{x}]$;
3. $\mathcal{F}[x, y], \mathcal{F}[x, \overline{y}]$, $\mathcal{F}[\overline{x}, y]$, $\mathcal{F}[\overline{x}, \overline{y}]$.

После построения классов формул в одном из перечисленных выше случаев программа применяет все правила упрощения (данные ей в качестве входных данных) ко всем полученным классам до тех пор, пока хотя бы одно из правил применимо. Правила применяются в порядке, в котором они даются программе, каждое правило применяется только тогда, когда ни одно из предыдущих правил не применимо. После этого программа вычисляет результирующее число расщепления относительно данной меры сложности.

### 3.4. Алгоритм.

Основной целью нашей программы является доказательство верхней оценки на время работы алгоритма, форма которого дана на рис. 1.

**Процедура АлгоритмРасщепления**

**Вход:**

формула $F$, множество правил упрощения $\mathcal{S}$, мера сложности $\mu$.

**Метод**

1. применять правила упрощения из множества $\mathcal{S}$ к формуле $F$ до тех пор, пока хотя бы одно из них применимо
2. если решение для $F$ может быть построено за полиномиальное время (в частности, если $F$ пуста), вернуть решение
3. для каждой пары переменных $x$ и $y$ рассмотреть следующие расщепления:
   (a) $F[x]$, $F[\overline{x}]$
   (b) $F[x, y]$, $F[x, \overline{y}]$, $F[\overline{x}]$
   (c) $F[x, y]$, $F[x, \overline{y}]$, $F[\overline{x}, y]$, $F[\overline{x}, \overline{y}]$

   выбрать то, которому соответствует максимальное число расщепления (относительно $\mu$)
4. построить формулы, соответствующие расщеплению, найденному на предыдущем шаге, и произвести рекурсивные вызовы для этих формул
5. вернуть ответ, основываясь на ответах, возвращенных рекурсивными вызовами

Рис. 1. Форма алгоритма расщепления.

Получив на вход множество правил упрощения $\mathcal{S}$, меру сложности $\mu$ и число *SplitNum*, наша программа пытается доказать, что такой алгоритм всегда находит расщепления с числом расщепления, не превосходящим *SplitNum*. Таким образом, она пытается доказать верхнюю оценку $O(SplitNum^{\mu})$ для такого алгоритма.

Основная часть нашей программы реализована в процедуре *АнализироватьСлучай*, которая представлена на рис. 2. Говоря неформально, эта процедура пытается найти подходящее расщепление для входного класса формул. Если такое расщепление найдено, программа возвращает, в противном случае она строит несколько новых классов, объединение которых содержит исходный класс, и рекурсивно вызывает себя на этих классах. Легко видеть, что если эта процедура останавливается на данном классе формул, то для каждой формулы этого класса есть подходящее расщепление.

---

**Процедура АнализироватьСлучай**

**Вход:**

класс формул $\mathcal{F}$, множество правил упрощения $\mathcal{S}$, число $SplitNum > 1$, мера сложности $\mu$.

**Метод**

1. если к $\mathcal{F}$ применимо хотя бы одно правило множества $\mathcal{S}$, вернуть
2. если для $\mathcal{F}$ есть расщепление относительно $\mu$, которому соответствует число расщепления, не превосходящее $SplitNum$, вернуть
3. выбрать из $\mathcal{F}$ согласно некоторой эвристике клоз $C$ с неопределенной длиной
4. (a) $C_1 = C$
   (b) фиксировать длину $C_1$
   (c) произвести рекурсивный вызов для $\mathcal{F} + \{C_1\} - \{C\}$
5. (a) ввести новую переменную $u$ в $\mathcal{F}$
   (b) $C = C + \{u\}$
   (c) для всех $i$, $j > 0$, таких что $(i, j) \in \alpha(\mu, SplitNum)$,
      i. построить множество всех возможных классов формул, получающихся из $\mathcal{F}$ добавлением вхождений переменной $u$, при которых $u$ становится $(i, j)$-переменной
      ii. произвести рекурсивный вызов на каждом построенном классе формул

---

Рис. 2. Процедура АнализироватьСлучай.

На первом шаге процедура *АнализироватьСлучай* проверяет, применимо ли к $\mathcal{F}$ хотя бы одно правило множества $\mathcal{S}$, и возвращает, если такое правило есть. Этот шаг мотивируется тем фактом, что алгоритм расщепления (описанной выше формы) расщепляет только упрощенные формулы. На втором шаге процедура пытается найти подходящее расщепление (то есть расщепление, которому соответствует число расщепления, не превосходящее $SplitNum$) для входного класса формул и возвращает, если такое расщепление есть (в подразделе 3.3 мы показываем, как именно программа делает это).

На третьем шаге процедура просто выбирает клоз с неопределенной длиной. Отметим, что ситуация, когда таких клозов нет, невозможна, поскольку в противном случае все известные переменные класса $\mathcal{F}$ образовывали бы замкнутую подформулу и, следовательно, было бы применимо соответствующее правило упрощения.

На четвертом и пятом шагах процедура рассматривает два случая: когда выбранный клоз содержит еще хотя бы один литерал и когда не содержит. В некотором смысле, она расщепляет класс формул, для которого не может найти подходящее расщепление, на несколько более мелких классов и рекурсивно вызывает себя на них. Целью данных двух шагов является построение классов формул, объединение которых содержит $\mathcal{F}$.

Таблица 2. Автоматически доказанные верхние оценки.

| | $N$ | $K$ | $L$ |
|---|---|---|---|
| SAT | $O(2.000000^N)$ | $O(1.272021^K)$ | $O(1.112777^L)$ |
| MAXSAT | $O(2.000000^N)$ | $O(1.372260^K)$ | $O(1.135889^L)$ |
| $(n,3)$ – MAXSAT | $O(1.285200^N)$ | $O(1.236507^K)$ | $O(1.098267^L)$ |

Мы используем отношение $\alpha$ на пятом шаге процедуры для обозначения множества пар $(i,j)$, таких что $(i,j)$-литерал не предоставляет немедленно подходящего расщепления. Легко видеть, что для любой задачи это отношение может быть определено следующим образом:

$$\alpha(K, SplitNum) = \{(i,j) : \tau(i,j) > SplitNum\},$$

$$\alpha(L, SplitNum) = \{(i,j) : \tau(i+j, i+j) > SplitNum\},$$

где $K$ – количество клозов входной формулы, $L$ – ее длина. Эти множества конечны вследствие свойств чисел расщепления [11].

К сожалению, мы не можем определить такое конечное отношение, когда работаем с числом переменных, как с мерой сложности. Тем не менее, в случае задачи $(n, 3)$-MAXSAT возможно написать следующее:

$$\alpha(N, SplitNum) = \{(i, j) : i + j \leq 3\},$$

где $N$ – число переменных формулы.

Отметим, что на третьем шаге рассматриваемая процедура может выбрать *любой* клоз с неопределенной длиной, на практике же размер получающегося анализа случаев (когда мы записываем все классы формул, обработанные процедурой) сильно зависит от выбранного клоза. Текущая версия программы выбирает клоз следующим образом: для каждого клоза с неопределенной длиной строятся классы формул, как это делается на четвертом и пятом шагах процедуры, и считается количество плохих классов среди них (здесь мы называем класс плохим, если он не удовлетворяет условиям первых двух шагов процедуры). В итоге выбирается клоз, которому соответствует минимальное количество плохих классов. В подавляющем большинстве ситуаций эта эвристика выбора клоза позволяла получать доказательства меньшего размера, чем при использовании других эвристик. Более того, с некоторыми эвристиками программа не может доказать оценку, в то время как эту оценку можно доказать с другой эвристикой. Таким образом, эвристика выбора клоза является очень важной частью алгоритма (см. также обсуждение в разделе 5).

**3.5. Автоматически доказанные верхние оценки.**

Верхние оценки, доказанные нашей программой, представлены в таблице 2. Выходные файлы с доказательствами доступны по адресу `http://logic.pdmi.ras.ru/~kulikov` (там же представлено несколько доказательств более простых оценок). Структура доказательств позволяет проверять каждый шаг программы. Отметим здесь, что большое количество случаев во всех представленных доказательствах не означает, что соответствующие алгоритмы сложны для реализации (на самом деле, форма таких алгоритмов дана на рисунке 1).

Таблица 3 содержит статистику рассматриваемых доказательств. Ниже мы вводим обозначения, использующиеся в таблице.

**оценка:** Задача и верхняя оценка для нее.

**время:** Время работы программы в секундах.

**случаи:** Количество случаев в доказательстве.

**уз. случаи:** Количество "узких" случаев (то есть случаев, в которых найденное число расщепления соответствует как раз ожидаемой верхней оценке).

**глубина:** Максимальная глубина рекурсии процедуры *АнализироватьСлучай*.

Таблица 3. Статистика.

| оценка | время | случаи | уз. случаи | глубина |
|---|---|---|---|---|
| $\mathrm{SAT}O(1.272021^K)$ | 1 | 40 | 25 | 2 |
| $\mathrm{SAT}O(1.112777^L)$ | 7 | 389 | 11 | 2 |
| $\mathrm{MAXSAT}O(1.372260^K)$ | 2632 | 9862 | 400 | 6 |
| $\mathrm{MAXSAT}O(1.135889^L)$ | 2968 | 19566 | 268 | 5 |
| $(n,3)-\mathrm{MAXSAT}O(1.285200^N)$ | 49637 | 16074 | 20901 | 8 |
| $(n,3)-\mathrm{MAXSAT}O(1.236507^K)$ | 923 | 3077 | 472 | 6 |
| $(n,3)-\mathrm{MAXSAT}O(1.098267^L)$ | 143 | 1257 | 2452 | 5 |

Несмотря на то, что лучшие известные на данный момент алгоритмы для SAT и MAXSAT используют метод расщепления, наша программа не смогла доказать их оценки. Мы полагаем, причиной тому является то, что эти алгоритмы содержат некоторые нестандартные шаги, которые трудно добавить в нашу программу (равно как и во все известные на данный момент подобные программы). Ниже мы кратко описываем эти шаги:

- Алгоритм Гирша для SAT ([9]) использует следующее правило упрощения: если каждый клоз формулы, который содержит $(2,3^+)$-литерал, содержит также и $(3^+,2)$-литерал, то всем $(3^+,2)$-литералам может быть присвоено значение `true`. Понятно, что такое правило невозможно добавить в нашу программу, поскольку программа может рассматривать только *константное* количество литералов.
- Алгоритм Чена и Кана для MAXSAT ([3]) сохраняет некоторый инвариант входной формулы. То есть он применяет правила упрощения или расщепляет входную формулу, только если все получающиеся формулы удовлетворяют

этому инварианту. Это делается для того, чтобы избежать некоторых "узких" случаев.

Приведенные оценки являются наилучшими из доказанных нашей программой для рассматриваемых задач. Как мы уже отмечали выше, невозможно заранее сказать, остановится ли наша программа на конкретных входных данных. Таким образом, мы были вынуждены прервать программу, когда она пыталась доказать более сильные оценки.

## 4. Подобные работы

Независимо от нас были написаны две программы для автоматических доказательств верхних оценок. Первая из них, написанная Николенко и Сироткиным [12], была разработана для доказательства верхних оценок для SAT и использует только одно правило упрощения, а именно: правило удаления чистых литералов. Этой программой было доказано, что задача SAT может быть решена за время $O(1.56639^K)$ алгоритмом расщепления, использующим только правило удаления чистых литералов. Другая программа была разработана Граммом и др. [7]. Этой программой было доказано несколько новых оценок для сложных задач модификации графов. Например, верхняя оценка для задачи Cluster Editing была улучшена до $O(1.92^k)$ (где $k$ обозначает количество допустимых модификаций ребер). Программа Грамма и др. позволяет работать с различными правилами упрощения.

Все современные программы (включая нашу) основаны на следующем простом замечании: возможно доказать верхнюю оценку для задачи, просто рассматривая все возможные подформулы размера, ограниченного некоторой константой. Программа Грамма и др. является, в некотором смысле, непосредственной реализацией этого факта: программа, получив на вход число $s$, строит множество $\mathcal{S}$ графов на $s$ вершинах, такое что любой возможный граф содержит в качестве подграфа хотя бы один элемент множества $\mathcal{S}$. Для каждого графа множества $\mathcal{S}$ перебираются все возможные расщепления и выбирается то, которому соответствует наименьшее число расщепления. Наконец, как результат, выдается оценка, соответствующая максимальному из рассмотренных чисел расщепления. Ясно, что итоговая оценка тем лучше, чем больше входное число $s$. Тем не менее, с увеличением числа $s$ очень сильно растет количество возможных расщеплений графа на $s$ вершинах (равно как и размер самого множества $\mathcal{S}$).

Ниже мы приводим отличия нашей программы от программы, написанной Граммом и др.

1. Наша программа сначала получает на вход оценку и после этого пытается ее доказать, в то время как программа Грамма и др. по данному числу $s$ пытается доказать как можно лучшую оценку, рассматривая графы на $s$ вершинах.
2. Множество возможных подформул в нашей программе строится в процессе поиска доказательства, построение же аналогичного множества в программе Грамма и др. делается на первом шаге. Таким образом, мы никогда не знаем, остановится ли (то есть докажет ли) наша программа данную оценку. Аналогично, невозможно сказать заранее, для какого числа $s$ (и существует ли такое $s$ вообще) программа Грамма и др. сможет доказать данную оценку.
3. Наша программа не рассматривает все возможные расщепления для каждого класса формул (она рассматривает лишь расщепления по наборам, содержащим не более, чем две переменные), как это делает программа Грамма и др.

## 5. Дальнейшие направления

Было бы интересно разработать эвристики быстрого нахождения хороших расщеплений для маленьких формул (или графов). Например, авторы работы [7] указывают на то, что в большинстве случаев почти все время (работы их программы) тратится на конкатенацию векторов расщепления.

Следующим направлением для дальнейших исследований является разработка эффективной процедуры построения множества всех возможных маленьких подформул (подграфов). Ясно, что такое множество далеко не единственно (к примеру, различные такие множества могут быть построены посредством выбора различных клозов на третьем шаге процедуры *АнализироватьСлучай*), и результирующая верхняя оценка зависит от этого множества.

Все известные на данный момент программы не предоставляют инструмента для конвертации выходных файлов в алгоритмы расщепления. Полученный алгоритм расщепления может использовать выходной файл программы как инструкцию. Таким образом, вполне возможно, что такой алгоритм мог бы работать лучше, чем простой алгоритм расщепления (поскольку ему не пришлось бы тратить время на нахождение хороших расщеплений).

Лучший известный на данный момент алгоритм для SAT [9] использует следующее правило упрощения: если каждый клоз формулы, содержащий $(2, 3^+)$-литерал, содержит также и $(3^+, 2)$-литерал, то всем $(3^+, 2)$-литералам может быть присвоено значение `true`. Видно, что такое правило не может быть добавлено ни в одну из известных программ, поскольку все они рассматривают только формулы (графы) ограниченного размера. Таким образом, было бы интересно ввести каким-либо образом это правило в существующие программы.

Другой нестандартный шаг используется лучшим известным алгоритмом для MAXSAT [3]: на каждой итерации алгоритм сохраняет некоторый инвариант входной формулы. Мы собираемся добавить такую возможность в нашу программу.

**Благодарности.**

Авторы признательны Эдуарду Алексеевичу Гиршу за привлечение их внимания к задаче. Мы также хотели бы поблагодарить анонимных рецензентов за множество полезных комментариев.

## Литература

1. J. M. Byskov, B. A. Madsen, and B. Skjernaa, *New Algorithms for Exact Satisfiability.* Theoretical Computer Science Preprint (2003).
2. N. Bansal and V. Raman, *Upper Bounds for MaxSat: Further Improved.* — Proceedings of ISAAC'99 (1999), pp. 247–258.
3. J. Chen and I. Kanj, *Improved exact algorithms for MAX-SAT.* To appear in Discrete Applied Mathematics.
4. М. А. Всемирнов, Э. А. Гирш, Е. Я. Данцин, С. В. Иванов, *Алгоритмы для пропозициональной выполнимости и верхние оценки их сложности.* — Зап. научн. семин. ПОМИ **277** (2001), 14–46.
5. M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem-proving.* — Comm. ACM **5** (1962), 394–397.
6. А. С. Куликов, С. С. Федин, *Решение задачи о максимальном сечении за время* $2^{|E|/4}$. — Зап. научн. семин. ПОМИ **293** (2002), 129–138.
7. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier, *Automated Generation of Search Tree Algorithms for Hard Graph Modification Problems.* — Algorithmica **39**, (4) (2004), 321–347.

8. J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith, *New worst-case upper bounds for MAX-2-SAT with application to MAX-CUT.* — Discrete Applied Mathematics **130**, (2), (2003) 139–155.
9. E. A. Hirsch, *New worst-case upper bounds for SAT.* — J. Automated Reasoning **24** (4) (2000), 397–420.
10. А. С. Куликов, *Верхняя оценка* $O(2^{0.16254n})$ *для* X3SAT*: более простое доказательство.* — Зап. научн. семин. ПОМИ **293** (2002), 118–128.
11. O. Kullmann and H. Luckhardt, *Algorithms for SAT/TAUT decision based on various measures.* — Informatics and Computation (1998).
12. S. I. Nikolenko and A. V. Sirotkin, *Worst-case upper bounds for SAT: automated proof.* — in: Proceedings of the 8th ESSLLI Student Session (2003), pp. 225–232.
13. V. Raman, B. Ravikumar, and S. Srinivasa Rao, *A Simplified NP-complete MAXSAT Problem.* — Information Processing Letters **65** (1998), 1–6.

Fedin S. S., Kulikov A. S. Automated proofs of upper bounds on the running time of splitting algorithms.

The splitting method is one of the most powerful and well-studied approaches for solving various NP-hard problems. The main idea of this method is to split an input instance of a problem into several simpler instances (further simplified by certain *simplification rules*), such that when the solution for each of them is found, one can construct the solution for the initial instance in polynomial time. There exists a huge number of articles describing algorithms of this type and usually a considerable part of such an article is devoted to case analysis. In this paper, we present a program that, given a set of simplification rules, automatically generates a proof of an upper bound on the running time of a splitting algorithm using these rules. As an example we report the results of experiments with such a program for the SAT, MAXSAT, and $(n, 3)$-MAXSAT (the MAXSAT problem for the case where every variable in the formula appears at most three times) problems.

Санкт-Петербургский
государственный университет

Поступило 11 октября 2004 г.