Санкт-Петербургский государственный университет

ТАГИН Евгений Юрьевич

Выпускная квалификационная работа

Алгоритмы для частных случаев задачи максимальной выполнимости

Образовательная программа магистратура “Математика”
Направление и код: 01.04.01 “Математика”
Шифр ОП: СВ.5832.2020 “Современная математика”

Научный руководитель:
доцент
Факультет математики
и компьютерных наук СПбГУ
кандидат ф.-м. наук
Близнец И.А.

Рецензент:
профессор
Джорджтаунский университет
кандидат ф.-м. наук
Головнёв А.Г.

Санкт-Петербург
2022 год

Оглавление

Глава 1

Введение

Задача выполнимости булевых формул (SAT) – одна из самых известных NP-трудных задач. В задаче требуется определить, существуют ли такие значения переменных, при которых эта формула выполняется. Её обобщение, задача максимальной выполнимости (MaxSAT), в которой требуется максимизировать количество выполненных клозов, также известна и поиск эффективных алгоритмов для неё и связанных с ней подзадач представляет большой интерес, как в теории, так и на практике ([1], [2], [3]).

Это одна из первых задач, для которой была показана NP-трудность. Помимо прочего исследовались подходы к

- различным подвидам `MaxSAT` ([4], [5])
- построению вероятностных и приближённых алгоритмов ([6], [7]),
- построению алгоритмов, использующих эвристики ([1]),
- построению точных алгоритмов ([8], [9], [10]), а также
- автоматическому построению доказательств верхних оценок на время работы точных алгоритмов ([11], [12], [13]).

Нас интересует последнее, а именно автоматическое построение точных алгоритмов. Типичный точный алгоритм, принимая на вход формулу F, на каждом шаге, покуда возможно, упрощает формулу правилами упрощения, а потом применяет правило расщепления: делает два рекурсивных вызова на подформулах полученных из F присваиваниями $l =$ `true` и $l =$ `false`, где l – переменная формулы F. После чего алгоритм возвращает ответ, исходя из ответов на подформулах. В общем случае в расщеплении может рассматриваться большее количество подформул и присваиваний.

Автоматическое построение точного алгоритма подразумевает построение

- алгоритма упрощения формулы;
- алгоритма поиска наилучшего расщепления;
- уточнения вида формулы (если не нашлось подходящих упрощений и расщеплений).

В последних двух работах именно такой метод автоматического построения алгоритмов и рассматривается:

- в работе [12] описывается программа, которая на основе заданного списка правил упрощения и ветвления строит для заданной формулы и константы расщепления точный алгоритм;
- в работе [13] исследуется подход к автоматической генерации правил упрощения.

На основе этих работ были получены новые оценки для `SAT` и `MaxSAT`.

Результаты, представленные в работе.

Цель данной работы состоит в развитии этого подхода. А именно в описании программы, который для заданного времени времени работы генерирует точный алгоритм для задачи выполнимости. Ключевыми особенностями алгоритма являются

- отсутствие необходимости явного описания большей части известных правил упрощения;
- построение правил упрощения на основе генерации возможных формул, а не подстановок в литералы значений (e.g. $x = 1$) или других литералов (e.g. $x = \overline{y}$);

На основе программы была улучшена верхняя оценка для частного случая `MaxSAT` – `(n,3)-MaxSAT`, в котором все переменные встречаются не более трёх раз, с $\mathcal{O}^*(1.191^n)$, относительно n, количества переменных в формуле, до $\mathcal{O}^*(1.175^n)$. Известные результаты по этой задаче представлены в таблице ниже.

n	Работа		Год
$\mathcal{O}^*(1.732^n)$	Раман, Равикумар, Рао	[14]	1998
$\mathcal{O}^*(1.325^n)$	Банзал, Раман	[8]	1999
$\mathcal{O}^*(1.273^n)$	Куликов	[15]	2009
$\mathcal{O}^*(1.260^n)$	Близнец	[16]	2012
$\mathcal{O}^*(1.237^n)$	Чен, Ксу, Уанг	[17]	2017
$\mathcal{O}^*(1.194^n)$	Ли, Ксу, Уанг, Янг	[18]	2017
$\mathcal{O}^*(1.191^n)$	Белова, Близнец	[5]	2020

Глава 2

Предварительные сведения

Назовём x и $\overline{x}$ *литералами* булевой переменной x. Будем называть литерал x *положительным*, а $\overline{x}$ – *отрицательным*. *Клозом* C называется дизъюнкция литералов, например, $C = (x \vee y \vee z) = (x\ y\ z)$. *Единичным* клозом будем называть клоз, состоящий ровно из одного литерала, например, $C = (\overline{x})$ *Длиной клоза* называется количество литералов в клозе (например, длина клоза $(x \vee \overline{y})$ – 2). *Юнит-клоз* – клоз длины 1. Формула в виде КНФ – конъюнкция клозов. *Длина формулы* (L) – количество литералов во всех клозах формулы. *Назначение* – функция, которая каждой переменной формулы ставит в соответствие 0 или 1. *Подназначение* – сужение назначения на некоторый набор переменных. Клоз называется выполненным (удовлетворённым) назначением, если назначение для какого-то из его литералов равно 1. Назначение *оптимальное*, если оно удовлетворяет максимальное количество клозов. *Сужение формулы* F на подназначение $I = \{x_i = v_i\}_{i=1}^{k}$ будем называть новую формулу, полученную заменой переменных x_i на v_i в F и обозначать как $F[I]$.

k-переменной будем называть переменную, которая встречается в формуле ровно k раз. *k^+ / k^--переменная* – переменная, встречающаяся в формуле k и более / менее раз. (i, j)-переменная – переменная x, такая что литерал x встречается в формуле i раз, а $\overline{x}$ – j раз. Нотации k, k^+, k^- будем называть *типом* переменной, нотацию (i, j) будем называть *литеральным типом* переменной.

Алгоритмы, строимые программой, основаны на технике расщепления ([19], [20]). Они задаются набором *правил упрощения*, используемых для упрощения исходной задачи, и *правил расщепления*, преобразующих задачу в несколько подзадач меньшего размера. Ответы для подзадач находится рекурсивно, из которых вычисляется ответ для исходной задачи. В общем случае подзадачи могут выглядеть произвольным обра-

зом, достаточно только уметь восстанавливать по ним ответ. Но мы будем рассматривать только расщепления, в которых для формулы F (задаче) и некоторого набора подназначений $I_1, \dots I_k$, сопоставляются подформулы $F[I_1], \dots, F[I_k]$ (подзадачи). Если задача размера n при применении некоторого правила расщепления расщепляется на k подзадач с размерами $n - t_1, \dots, n - t_k$, то вектор $(t_1, \dots, t_k)$ будем называть *вектором расщепления* для данного правила расщепления. *Константой расщепления* для этого правила будем называть положительный корень многочлена $x^n = x^{n-t_1} + \dots + x^{n-t_k}$. Если алгоритм использует правила расщепления с константами $c_1, \dots, c_t$, то время работы этого алгоритма не превышает $\mathcal{O}^*((\max_{i=1}^{t} c_i)^n)$.

Глава 3

Алгоритм

Алгоритм расщепления сначала упрощает, а потом расщепляет полученную на вход формулу F. Для генерации такого алгоритма мы будем рассматривать дерево формул с F в корне, внутренние узлы которого будут уточнениями формулы из родительского узла, а листья – формулы, на которых применимы правила упрощения и расщепления.

Структуры данных и реализация

Формула представляется программой как описание клозов и контекста, содержащего атрибуты элементов, входящих в клозы. Клоз представляется списком из

- литералов (e.g. $x, \overline{y}$),
- подклозов (e.g. A), а также
- *необлитерованных переменных* (e.g. z^θ) – литералов, соответствующих некой переменной, для которых не определено, как именно он входит в клоз, положительно или отрицательно. Например, клоз (xz^θ) означает (xz) или $(x\overline{z})$.

Также мы будем называть *подклоз-группой* всё содержимое клоза кроме литералов.

Контекстные атрибуты бывают двух типов:
для переменных их тип (e.g. 4^+-переменная, 5-переменная, $(3,2)$-переменная)
для подклозов запрет на наличие каких-то переменных (e.g. $x \notin C$)

В большинстве случаев с описанными выше формулами можно работать также как и с обычными формулами.

Программа принимает на вход формулу (e.g. $(xA)(xB)(\overline{x}C)$), после чего строит дерево разбора этой формулы, в корне которого расположена формула, а в остальных узлах – подвиды этой формулы. При этом

- дети **внутреннего** узла с формулой F – набор подвидов $\{F_1, \ldots, F_k\}$, такой что $F_1 \sqcup \cdots \sqcup F_k = F$. Их построение описывается в подразделе Генерация правил уточнения;
- в листьях помимо формулы содержатся правила упрощения или правила ветвления для соответствующей формулы.

Генерация правил уточнения

В программе рассматриваются уточнения двух типов:

- Уточнение подклоза Мы рассматриваем все возможные варианты того, как выглядит подклоз – он
  - либо пустой;
  - либо содержит уже присутствующую в формуле переменную;
  - либо содержит новую переменную.

  Пример 1: для формулы $(x^\theta A)(y^\theta B)$ после уточнения A получатся уточнённые формулы
  - $(x^\theta)(y^\theta B)$,
  - $(x^\theta y^\theta A)(y^\theta B)$,
  - $(x^\theta w^\theta A)(y^\theta B)$, причём $y^\theta \notin A$ (в противном случае мы бы рассмотрели такой вариант в предыдущем уточнении)

  Также при добавлении присутствующей переменной не рассматриваются переменные,
  - которые уже входят в формулу в количестве, соответствующем их типу (e.g. у 2-переменной не может быть больше 2-х вхождений);
  - уже присутствующие в клозе с раскрываемым подклозом – формулу такого вида можно было бы сразу упростить (подробнее это описано в разделе 3).

  Добавляя новую переменную в подклоз, мы также уточняем её тип в контексте формулы – 3^+ и t^-, где t – максимальное количество вхождений какой-либо из переменных. 1- и 2-переменные не рассматриваются, потому что формулы с ними также упрощаемы. Верхняя граница на количество вхождений переменной выставляется для упрощения перебора – мы считаем, что максимальное количество вхождений переменных в рассматриваемых формулах не превышает максимального количества вхождений переменных в формуле, поданной программе на вход.
- Уточнение переменной Уточнения переменной бывают 4-х видов:
  1. Тип переменной (e.g. $3^+ \to \{3, 4^+\}$)
  2. Литеральный тип переменной (e.g. $4 \to \{(3,1),(2,2)\}$).

3. Позиции переменных в клозах.
4. Значения литералов переменных (*облитерация*).

Пример 2: для формулы $(y^\theta A)(B)$, где y – 3-переменная, после уточнения позиции y получатся
- $(y^\theta A)(By^\theta)(Cy^\theta)$,
- $(y^\theta A)(B)(Cy^\theta)(Dy^\theta)$.

Пример 3: для формулы $(y^\theta A)(By^\theta)(Cy^\theta)$ после уточнения значений литералов y получатся
- $(yA)(By)(C\overline{y})$
- $(yA)(B\overline{y})(Cy)$
- $(\overline{y}A)(By)(Cy)$

Каждый вид уточнения переменной производится только после уточнения предыдущих видов. Это позволяет
- рассматривать перебор только литеральных типов (i, j) вида $i \geq j$ (*как следствие, 3-переменные могут быть только* $(2,1)$*-переменными, но в контексте при этом литеральный тип указывается только после облитерации*),
- считать, что в подклоз-группах не содержится литералов, соответствующим облитерованным переменным.

Таблица назначений

Пусть $\mathcal{A}$ – множество назначений облитерованным переменным в F, $a \in \mathcal{A}$, а *домен*, Δ – множество подклоз-групп формулы. Тогда назовём *выигрышем клозов* функцию $\mathrm{cgain}(F)[s]$, показывающую, сколько в F будет выполнено клозов в зависимости от значений Δ. Выигрыш cg состоит из двух частей:
- $\mathrm{num}(cg) \in \mathbb{N}$ – гарантированно выполненное количество клозов;
- $\mathrm{symbols}(cg)$ – элементы подклоз-группы.

Например, $\mathrm{cgain}((xA)(\overline{x}B))[x = 0] = A{+}1$. A в правой части равенства подразумевает значение, которое принимает подклоз (0 или 1).

Таблицей назначений для данной формулы мы будем называть таблицу, в первой колонке которой перечисляются все возможные назначения облитерованных переменных из формулы, а элементы второй колонки соответствуют значениям cgain на данных назначениях.

Пример, для формулы $F = (x)(xB)(\overline{x}C)$ таблица назначений выглядит следующим образом:

x	cgain
0	$1 + B$
1	$2 + C$

Для выигрыша клозов c и подстановки $\delta : \Delta \to \{0, 1\}$ определено значение $c[\delta]$, получаемое подстановкой значений домена в выигрыш клозов – это точное значение клозов, выполненных в формуле.

Пусть F – формула, а $s, t \in \mathcal{A}$ – 2 назначения для F. Будем говорить, что выигрыш $\text{cgain}(F, s)$ *мажорирует* выигрыш $\text{cgain}(F, t)$, если $\forall \delta \in \Delta \quad \text{cgain}(F, s)[\delta] \geq \text{cgain}(F, t)[\delta]$. *Мажорирующие выигрыши* формулы – множество выигрышей, которые не мажорируют другие выигрыши. Соответственно, *мажорирующим набором назначений* будем называть мажорирующие назначения, соответствующие всем мажорирующим выигрышам. Например, в таблице выше назначение $x = 1$ мажорирует назначение $x = 0$, выигрыш $2 + C$ – мажорирующий. *Мажорирующим набором назначений* будем называть набор назначений, которому соответствуют все мажорирующие выигрыши формулы. Далее в таблицах выигрышей формул мы будем дополнительно добавлять колонку `"m"` с указанием номеров у мажорирующих выигрышей – мы можем считать, что они как-то упорядочены.

Пример:

x	cgain	m
0	$1 + B$	
1	$2 + C$	1

Описание алгоритма построения мажорирующих выигрышей представлено в таблице 3.1.

Генерация правил упрощения

Для упрощения мы будем перебирать мы воспользуемся тем соображением, что если у двух формул F и F_1 одинаковые мажорирующие выигрыши MC, то по оптимальному ответу для одной можно найти оптимальный ответ для другой – для этого достаточно в F выбрать назначение, соответствующее мажорирующему выигрышу оптимального назначения F_1.

Для поиска упрощённой формулы мы сначала генерируем таблицу назначений, извлекаем из неё мажорирующие выигрыши клозов, после этого перебираем все возможные формулы, такие что для неё мажорирующие

Алгоритм: `cgain_dominated_by`(a, b)

Вход: a, b – выигрыши клозов

Вывод: мажорирует ли выигрыш a выигрыш b

$\text{anum}, \text{asymb} \leftarrow \text{num}(a), \text{symbols}(a)$
$\text{bnum}, \text{bsymb} \leftarrow \text{num}(b), \text{symbols}(b)$
вернуть $\text{anum} \geq \text{bnum} + |\text{bsymb} \setminus \text{asymb}|$

Алгоритм: `find_major_cgains`(G)

Вход: Формула F

Вывод: Мажорирующие назначения и выигрыши F

$\mathcal{S} \leftarrow$ подстановки F
$\text{cgains} \leftarrow$ выигрыши клозов в F на подстановках $\mathcal{S}$
$\text{majorCgains} \leftarrow \varnothing$
для каждого $c \in \text{cgains}$ **выполнить**
  если $\nexists c' \in \text{cgains} : \text{cgain\_dominated\_by}(c, c')$ **то**
    $\text{majorCgains} \leftarrow \text{majorCgains} \cup \{c\}$
вернуть majorCgains

Рис. 3.1: Алгоритм поиска мажорирующих выигрышей

выигрыши такие же, как и у исходной формулы. После этого остаётся выбрать среди них формулу минимальной длины. Если она меньше исходной формулы, то мы считаем, что упрощение нашлось.

Подробное описание алгоритма представлено в таблице 3.2.

В алгоритме используется ряд процедур, не описанных ранее:

- процедура `asgns` возвращает список всевозможных назначений на заданных переменных;
- процедура `formula` строит формулу по набору клозов;
- процедура `size` возвращает размер поданной на вход формулы. В нашем случае это L – общее количество литералов в формуле, но в общем случае это может быть и количество переменных, и количество клозов;
- процедура `find_literals` описана в таблице 3.3: она возвращает все возможные наборы литералов, а также `false` (пустой набор литералов) и `true`.

Заметим, что при работе с частными случаями формул, мы помимо соответствия MC проверяем, что формула принадлежит тому же классу формул, что и F. Например, для `(n,3)-MaxSAT` формулы мы проверяем,

Алгоритм: `generate_simplified_formula`(F)
Вход: Формула F
Вывод: Формула F' меньшего размера с такими же мажорирующими выигрышами

```
MC ← find_major_cgains(F); m ← |MC|
S ← упорядоченные элементы Δ из MC; s ← |S|
для каждого i ∈ [1 : m] выполнить
    upper[i] ← num(MC_i) + |symbols(MC_i)|
t ← max_i upper[i] − s
для каждого i ∈ [1 : m] выполнить
    для каждого j ∈ [1 : s] выполнить
        cval_j[i] ← 𝟙_{Δ_1[i] ∉ MC_i}
    для каждого j ∈ [s + 1 : s + t] выполнить
        cval_j[i] ← 𝟙_{j ≤ s + upper[i]}
F_1 ← null
k ← ⌈log_2 m⌉
для каждого B_m ⊆ asgns({a_1, ..., a_k}) : |B_m| = m выполнить
    для каждого j ∈ [1 : s + t] выполнить
        cvs_j ← generate_literals(B_m, cval_j)
    для каждого lits ∈ cvs_1 × ··· × cvs_{s+t} выполнить
        для каждого j ∈ [1 : s] выполнить
            C_j ← lits_j ∪ {Δ_1[j]}
        для каждого j ∈ [s + 1 : s + t] выполнить
            C_j ← lits_j
        F_res = formula(C_1, ..., C_{s+t})
        если find_major_cgains(F_res) = MC то
            если F_1 = null ∨ size(F_res) < size(F_1) то
                F_1 ← F_res
вернуть F_1
```

Рис. 3.2: Алгоритм построения упрощённой формулы

что каждая переменная в F_{res} входит не более трёх раз.

Заметим, что мы можем сразу дополнительно фильтровать наборы литералов, оставляя только содержащие минимальные по включению множества литералов. Например, если подходит наборы литералов $\{xy, x\}$, то

Алгоритм: `generate_literals`(assignments, clause_values)
Вход: набор назначений и значения клоза на этих назначениях
Вывод: наборы литералов, удовлетворяющих значениям клоза

res $\leftarrow \varnothing$
для каждого
lits $\in$ всевозможные литералы на переменных из assignments
выполнить
  если $\forall\ i$ lits[assignments$_i$] = clause_values$_i$ **то**
    res $\leftarrow$ res $\cup$ {lits}
вернуть res

Рис. 3.3: Алгоритм построения наборов литералов по значениям

xy мы включать не будем. Аналогично, если подходит `true`, то никакие другие группы литералов мы включать не будем.

- Пример 1: Пусть дана формула $F = (xAy)(xBy)(\overline{x}C)(\overline{y}D)$, где x и y – $(2,1)$-переменные.
  (1). Строим таблицу выигрышей клозов:

x y	cgain	m
0 0	$2+A+B$	1
0 1	$3+D$	3
1 0	$3+C$	2
1 1	$2+C+D$	

  Находим выигрыши: MC $\leftarrow \{2+A+B, 3+D, 3+C\}$
  (2). Строим таблицу cval – значения литералов из клозов C_j, соответствующих элементам Δ:

$i \setminus j$	$1(A)$	$2(B)$	$3(C)$	$4(D)$
$1(2+A+B)$	0	0	1	1
$2(3+C)$	1	1	0	1
$3(3+D)$	1	1	1	0

  (3). Перебираем наборы назначений $\{a_1, a_2, a_3\}$ и генерируем наборы подходящих литералов cvs. Допустим, мы сейчас рассматриваем $B_3 = [\{a_1a_2 = 01, a_1a_2 = 10, a_1a_2 = 11\}]$. Тогда

a_1	a_2	A	B	C	D
0	1	0	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

(4). Для каждого клоза перебираем множества групп литералов, удовлетворяющих значениям таблицы. Например:

j	cvs
1	$a_1, a_1\overline{a_2}$
2	$a_1, a_1\overline{a_2}$
3	$a_2, \overline{a_1}a_2$
4	$\overline{a_1}\ \overline{a_2}$

(5). Итоговая формула получается дополнением клозов элементами Δ_1:

$$F_1 = (a_1 A)(a_2 B)(a_2 C)(\overline{a_1}\ \overline{a_2} D)$$

(6). Проверяем, что у новой формулы такие же мажорирующие выигрыши клозов:

a_1	a_2	cgain	m
0	0	$1 + A + B + C$	
0	1	$2 + A + B$	1
1	0	$3 + C$	2
1	1	$3 + D$	3

(7). Размер новой формулы меньше: $|F_1| = 5 < 6 = |F|$ – формула подходит.

- Пример 2: Пусть дана формула $F = (\overline{x}\ \overline{y})(xz)(xw^{\theta})(yz)(yB)(\overline{z}C)F'$, где x и y – $(2,1)$-переменные, а w – 3-переменная.
(1) Строим таблицу выигрышей клозов:

x	y	z	cgain	m
0	0	0	$2 + B + w1$	
0	0	1	$3 + B + C + w1$	
0	1	0	$4 + w1$	
0	1	1	$4 + C + w1$	2
1	0	0	$4 + B$	
1	0	1	$4 + B + C$	1
1	1	0	5	3
1	1	1	$4 + C$	

Находим выигрыши: MC $\leftarrow \{4 + B + C, 4 + C + w1, 5\}$
(2). Строим таблицу cval – значения литералов из клозов C_j, соответствующих элементам Δ:

$i \setminus j$	$1(B)$	$2(C)$	$3(w1)$	4	5	6
$1(4+B+C)$	0	0	1	1	1	1
$2(4+C+w1)$	1	0	0	1	1	1
$3(5)$	1	1	1	0	1	1

(3). Перебираем наборы назначений $\{a_1, a_2, a_3\}$ и генерируем наборы подходящих литералов cvs. Допустим, мы сейчас рассматриваем $B_3 = [\{a_1a_2 = 00, a_1a_2 = 10, a_1a_2 = 11\}]$. Тогда

a_1	a_2	$1(B)$	$2(C)$	$3(w1)$	4	5	6
0	0	0	0	1	1	1	1
1	0	1	0	0	1	1	1
1	1	1	1	1	0	1	1

(4). `generate_literals` сгенерирует следующие группы литералов:

j	cvs
1	a_1
2	a_2
3	$\overline{a_1}a_2$
4	$\overline{a_2}$
5	`true`, $a_1\overline{a_2}$
6	`true`, $a_1\overline{a_2}$

(5). Итоговая формула получается дополнением клозов элементами Δ_1:

$$F_1 = (a_1B)(a_2C)(\overline{a_1}a_2w^\theta)(\overline{a_2})(\texttt{true})(\texttt{true})$$

(6). Проверяем, что у новой формулы такие же мажорирующие выигрыши клозов:

a_1	a_2	cgain	m
0	0	$4+B+C$	1
0	1	$4+B$	
1	0	$4+C+w1$	2
1	1	5	3

(7). Размер новой формулы меньше: $|F_1| = 5 < 6 = |F|$ – формула подходит.

Генерация покрывающих подназначений

Будем говорить, что подназначение s *покрывает* назначение a, если переменные s означены также, как и в a. Например, $\{x = 1\}$ покрывает $\{x = 1, y = 0\}$. Будем говорить, что набор подназначений S *покрывает* набор назначений A (или является *покрытием* A), если $\forall a \in A\ \exists s \in S$: s покрывает a.

Поиск покрытий мы производим следующим образом: если у назначений есть общее покрытие, то возвращаем его, а если нет, то для каждой переменной v разделяем назначения на две группы: покрываемые подназначениями $\{v = 0\}$ и $\{v = 1\}$. После этого рекурсивно ищем покрытие на получившихся группах. Описание алгоритма представлено в таблице 3.4.

Генерация правил упрощения и расщепления из покрытий

Расщепление состоит из набора покрытий назначений формулы. Заметим, что покрывать достаточно только мажорирующие наборы назначений. Формализуется это следующим образом: покрытие $S = \{s_1, \ldots, s_d\}$ *корректное* для формулы F, если

$$\forall \delta \in \Delta' \quad \max_{a \in \mathcal{A}} \operatorname{cgain}(F)[a][\delta] = \max_{i=1}^{d} \max_{a' \in \mathcal{A}[s_i]} \operatorname{cgain}(F)[a'][\delta]$$

Покрытия, состоящие из одного элемента, мы будем называть *1-покрытием* – если таковое нашлось, то мы можем упростить формулу, не прибегая к расщеплению. Поэтому их мы искать будем в первую очередь. Описание алгоритма поиска покрытий представлено в таблице 3.5.

- Улучшение правил расщепления Мы можем зафиксировать порядок вычисления ответа на подформулах, суженных на подназначения из расщепления. Тогда при вычислении ответа на очередной подформуле мы можем отфильтровать из набора пары назначений, у которых выигрыш не превышает выигрыши пар, рассмотренных в предыдущих подформулах. Это позволит сузить формулы в том числе и на необлитерованные переменные.
  В качестве иллюстрации рассмотрим формулу $F = (\overline{x})(\overline{x}y^{\theta})(xa^{\theta})(xw^{\theta})F'$.
  У неё два мажорирующих назначения:

x	cgain	m
0	$2 + a2 + w3$	1
1	$2 + y1$	2

Алгоритм: `common_subasgn`(A)

Вход: Набор назначений

Вывод: Подназначение, покрывающее назначения

res $\leftarrow \varnothing$
для каждого $v \in \text{vars}(A)$ **выполнить**
    res $\leftarrow$ res $\cup \{\{v = 0\} \mid v \in \text{vars}(A) : \forall\ a \in A\ \ \{v = 0\}$ покрывает $a\}$
    res $\leftarrow$ res $\cup \{\{v = 1\} \mid v \in \text{vars}(A) : \forall\ a \in A\ \ \{v = 1\}$ покрывает $a\}$
вернуть res

Алгоритм: `gen_covers`(A, d)

Вход: Набор назначений и ожидаемое количество покрытий

Вывод: Наборы подназначений размера $\leq d$, покрывающих назначения

$A_c \leftarrow$ `common_subasgn`(A)
если $A_c \neq \varnothing \vee d = 1$ **то**
    вернуть A_c
res $\leftarrow \varnothing$
для каждого $v \in \text{vars}(A)$ **выполнить**
    для каждого $i \geq 1, j \geq 1 : i + j = d$ **выполнить**
        $A_0 \leftarrow$ `gen_covers`$(\{a \in A \mid \{v = 0\}$ покрывает $a\}, i)$
        $A_1 \leftarrow$ `gen_covers`$(\{a \in A \mid \{v = 1\}$ покрывает $a\}, j)$
        для каждого $(S_0, S_1) \in A_0 \times A_1$ **выполнить**
            cover $\leftarrow \{s_0 \cup \{v = 0\} \mid s_0 \in S_0\} \cup \{s_1 \cup \{v = 1\} \mid s_1 \in S_1\}$
            res $\leftarrow$ res $\cup \{\{s \cup A_c \mid s \in \text{cover}\}\}$
вернуть res

Рис. 3.4: Алгоритм генерации покрывающих подназначений

Если мы сначала посчитаем ответ для формулы $F[x = 0]$, то в подсчёте ответа для $F[x = 1]$ нет смысла рассматривать случаи, когда $y1 = 0$: выигрыш в таком случае будет 2, что доминируется выигрышем $2 + a2 + w3$. И итоговый набор назначений – $[\{x = 0\}, \{x = 1, y1 = 1\}]$.

Обновлённый алгоритм `gen_covers` помимо набора назначений принимает набор выигрышей, соответствующих назначениям, а на эта-

Алгоритм: `generate_covers`(F)
Вход: Формула F
Вывод: Наборы расщеплений

covers $\leftarrow \varnothing$
для каждого $A \in$ мажорирующие наборы расщеплений F
выполнить
  $c_1 =$ `gen_covers`$(A, 1)$
  если $c_1 \neq \varnothing$ **то**
    вернуть c_1
  covers $\leftarrow$ covers $\cup$ `gen_covers`$(A, 2) \cup$ `gen_covers`$(A, 3)$
вернуть covers

Рис. 3.5: Алгоритм генерации покрытий

пе построения общего подназначения в `common_subasgn` добавляет означивания вида $\{y1 = 1\}$, если при $\{y1 = 0\}$ выигрыши мажорируются ранее рассмотренными выигрышами.

Связь с существующими правилами упрощения

Рассмотрим правила упрощения, рассматриваемых ранее в статьях [18], [5], [10]. Приведём их в табличной форме:

№	F	F_1
1	$(x \vee x \vee C)F'$	$(x \vee C)F'$
2	$(x \vee \overline{x} \vee C)F'$	$(\mathtt{true})F'$
3	$(x \vee X)(x \vee B)(x \vee C)F'$	$(\mathtt{true})(\mathtt{true})(\mathtt{true})F'$
4	$(x \vee A)(\overline{x} \vee B)F'$	$(A \vee B)(\mathtt{true})F'$
5	$(x)(x \vee A)(\overline{x} \vee B)F'$	$(\mathtt{true})(\mathtt{true})(B)F'$
6	$(x \vee y)(\overline{x})(\overline{y})F'$	$(\overline{x} \vee \overline{y})(\mathtt{true})F'$

- формулы вида (1)..(2) упрощаются алгоритмом `generate_simplified_formula`, поэтому для сокращения перебора в правилах уточнения такие формулы не рассматриваются
- для формул вида (3) найдётся 1-покрытие, поэтому их мы аналогично не рассматриваем в правилах уточнения
- формулы вида (5)..(6) также упрощаются `generate_simplified_formula`

Остаются формулы вида (4). Для них в программе реализована отдельная процедура.

Алгоритм: `generate_cases_tree`(F, c)
Вход: Формула F, желаемая константа расщепления c
Вывод: Дерево разбора F

$c_1 \leftarrow$ `gen_covers`$(A, 1)$
если $c_1 \neq \varnothing$ **то**
  вернуть Leaf$(F, \text{one_cover} = c_1)$
$F' \leftarrow$ `generate_simplified_formula`(F) **если** $F' \neq \text{null}$ **то**
  вернуть Leaf$(F, \text{simplified_formula} = F')$
$S \leftarrow$ `generate_covers`(F)
для каждого $s \in S$ **выполнить**
  $Fs \leftarrow \{F[s_i] \mid s_i \in s\}$
  $Fs' \leftarrow$ `generate_simplified_formula`(F)
  $t \leftarrow \{\texttt{size}(F) - \texttt{size}(F') \mid F' \in Fs'\}$
  $c' \leftarrow$ константа расщепления для t
  если $c' \leq c$ **то**
    вернуть Leaf$(F, \text{split} = s)$
$Fs \leftarrow$ уточнения F, где уточнение выбрано согласно некоторой эвристике
children $\leftarrow \{$`generate_cases_tree`$(F', c) \mid F' \in Fs\}$
вернуть InnerNode$(F, \text{children})$

Рис. 3.6: Алгоритм генерации дерева разбора

Алгоритм генерации дерева разбора

Осталось собрать всё вместе – алгоритму подаётся на вход формула F и желаемая константа расщепления c, на выходе генерируется дерево разбора этой формулы, причём в листьях присутствуют либо

- правила упрощения, представляемые 1-покрытием или упрощённой формулой с теми же мажорирующими выигрышами, что и у исходной формулы, либо
- правила расщепления, представляемые покрытием, расщепление по которому даёт константу $c' < c$.

Алгоритм представлен в таблице 3.6.

Эвристики выбора правила уточнения

Правило уточнения может быть выбрано для любого подклоза или необлитерованной переменной. Если таковых нет, то формула замкнутая, сле-

Алгоритм: `formulas_score`(Fs, c)
Вход: Набор формул Fs, желаемая константа расщепления c
Вывод: оценка трудоёмости разбора Fs

$Fs' \leftarrow \{F \in Fs \mid \texttt{generate\_cases\_tree}(F, c) \neq \text{Leaf}(\cdot, \cdot)\}$
$C \leftarrow \{$лучшие константы расщепления для $Fs'\}$
score $\leftarrow \sum\{(c' - c) \cdot \exp(\texttt{size}'(F)) \mid (F, c) -$ пары из $(Fs', C)\}$
вернуть score

Рис. 3.7: Алгоритм генерации дерева разбора

довательно найдётся лишь один мажорирующий выигрыш, и процедура завершит выполнение на этапе поиска 1-покрытия.

Размер получающегося дерева сильно зависит от выбора уточнения. На практике лучше всего показала себя эвристика, которая

- генерирует наборы уточнённых формул для всех правил уточнения,
- для всех уточнёнnных формул находит лучшие константы расщепления c_i для формул, оценивает их близость к c и оценивает трудоёмкость (оценка на время разбора) формул согласно процедуре из таблицы 3.7,
- выбирает уточнение с минимальной оценкой соответствующим ему формулам.

Процедура $\texttt{size}'$ похожа на `size` за тем исключением, что дополнительно штрафуются необлитерованные переменные x^θ – чем больше номер первого клоза с x^θ, тем больше штраф:

$$\texttt{size}'(F) = |\text{vars}(F)| + \sum\{1 + (\text{№ первого клоза с } x)/20 \mid x \in \texttt{unlit}(F)\}$$

, где $\texttt{unlit}(F)$ – необлитерованные переменные F.

Выбор такой эвристики обусловлен следующими соображениями:

- чем ближе константа расщепления для формулы к c, тем лучше;
- штраф за необлитерованные переменные помогает избегать ситуаций, когда в формуле вообще нет клозов без подклозов и необлитерованных переменных;
- время работы процедуры `generate_cases_tree`, не учитывая время рекурсивные вызовы, растёт экспоненциально относительно количества переменных.

Проверка корректности программы

Корректность большинства описанных выше алгоритмов легко проверить, как с помощью написания несложных тестов, так и визуальной оценки. Чего нельзя сказать об улучшенных правилах расщепления – беглое изучение таблицы выигрышей не позволяет сразу понять, корректно ли означивать необлитерованные переменные и точно ли покрываются все мажорирующие выигрыши. В свете этого для них было реализовано дополнительное тестирование на основе SMT-солвера `Z3`.

SMT (*satisfiability modulo theories*) – расширения задачи `SAT` новыми теориями, такими как теории целых и вещественных чисел, логики первого порядка и т.д. Нас будут интересовать логики и теории целых чисел.

Для проверки корректности покрытия формулы F набором подназначений S строится явное выражение $\mathrm{cgain}(F)$ и проверяется выражение, формально описывающее корректность покрытия:

$$\forall a \in \mathcal{A}, \delta \in \Delta' \; \exists a' \in \mathcal{A} : \texttt{covered}(a', \delta, S) \wedge \mathrm{cgain}(F)[a'][\delta] \geq \mathrm{cgain}(F)[a][\delta]$$

Процедура `covered` проверяет, что в S найдётся подназначение покрывающее a' и δ.

Так как нас интересует контрпример к корректности покрытия, мы рассматриваем отрицание выражения, проверяющего корректность:

$$\exists a \in \mathcal{A}, \delta \in \Delta' \; \forall a' \in \mathcal{A} : \neg\texttt{covered}(a', \delta, S) \vee \mathrm{cgain}(F)[a'][\delta] < \mathrm{cgain}(F)[a][\delta]$$

Если таких a и δ не нашлось, то покрытие корректное.

Рассмотрим пример. Для вышеописанной формулы $F = (\overline{x})(\overline{x}y^{\theta})(xa^{\theta})(xw^{\theta})F'$ и покрытия $[\{x = 0\}, \{x = 1, y1 = 1\}]$ строится программа

```
(set-logic LIA)
(define-fun clz ((a Bool)) Int (ite (or a) 1 0))
(define-fun clz ((a Bool) (b Bool)) Int (ite (or a b) 1 0))

(declare-const w3 Bool)
(declare-const a2 Bool)
(declare-const x Bool)
(declare-const y1 Bool)

(define-fun fcg
  ((w3 Bool) (a2 Bool) (x Bool) (y1 Bool))
  Int
```

```
  (+ (clz (not x)) (clz (not x) y1) (clz x a2) (clz x w3)))

(assert
  (forall
    ((x00 Bool))
    (or
    (not (or (= x00 false) (and (= y1 true) (= x00 true))))
    (not (<= (fcg w3 a2 x y1) (fcg w3 a2 x00 y1))))))
(check-sat)
(exit)
```

Глава 4

Автоматически доказанные верхние оценки

`(n,3)-MaxSAT` по мере L

В алгоритме, описанном выше, подразумевалось, что длина формулы оценивается через L – общее количество литералов в формуле. Мы можем считать, что формуле нет $1-$ и $2-$ переменных, иначе бы формула была бы сразу упрощена. Это означает, что через оценки для L считаются оценки для n: $\mathcal{O}^*(c^n) = \mathcal{O}^*(c^{L/3})$. Для построения алгоритма нам понадобится лемма из статьи [16]:

Лемма 1. *Если все отрицательные литералы входят только в единичные клозы, то задача решается полиномиальным алгоритмом.*

Из леммы следует, что вместо рассмотрения формулы вида $F_0 = (\overline{x}C)(xA)(xB)F'$ достаточно рассматривать только формулы вида $F = (\overline{x}y^\theta C)(xA)(xB)F'$. Для данной формулы был сгенерирован ряд деревьев разбора для различных констант ветвления. Из этого следует утверждение:

Утверждение 1. *Алгоритм* `generate_cases_tree` *на входе* $F = (\overline{x}y^\theta C)(xA)(xB)F'$ *и* $c = 1.05519$ *останавливается и строит дерево разбора, соответствующее точному алгоритму со временем работы* $\mathcal{O}^*(1.05519^L)$, *а также* $\mathcal{O}^*(1.175^n)$.

Файлы с доказательствами доступны по адресу https://github.com/evjava/n3-maxsat-inference-trees.
Статистика по различным деревьям разбора представлены в таблице:

Таблица 4.1: Статистика

константа (n)	константа (L)	вектор	# узл.	# лист.	# уз. случ.	глуб.
1.175	1.05519	(30 24 12)	81427	47080	29	48
1.179	1.05625	(33 21 12)	64134	36958	232	48
1.180	1.05664	(21 21 18)	49486	28454	453	47
1.184	1.05766	(24 21 15)	39516	22488	336	44
1.185	1.05793	(30 21 12)	36766	20925	164	44
1.186	1.05828	(30 27 9)	27344	15274	112	35
1.190	1.05947	(12 12)	21898	12413	403	34
1.191	1.05992	(21 6)	16450	9171	293	31
1.194	1.06084	(15 9)	1002	547	53	18
1.260	1.08006	(9 9)	146	79	5	9

- Важность леммы Лучшая оценка, которую получалось достичь для общего случая – $\mathcal{O}^*(1.08213^L)$. (соответствующий вектор ветвления – (15 15 12)). Достигается на случаях, когда в формуле большое количество негативных юнит-клозов.

MaxSAT по мере L

Была получена новая оценка для формул с 5^+-переменными: $\mathcal{O}^*(1.090508^L)$ (соответствующий вектор ветвления – (8 8)). Предыдущий результат – $\mathcal{O}^*(1.092639^L)$ (соответствующий вектор ветвления – (10 6)), получен в [10].

Но при этом 4-переменные представляют на данный момент нерешаемую проблему – попытки вывести дерево разбора для константы ветвления 1.091198 (соответствующий вектор ветвления – (14 4)) закончились неудачно.

Другие меры

На данный момент программа реализована только для нахождения оценок по мере L (и, как следствие, по мере n для `(n,3)-MaxSAT`). Но подход обобщаем и на другие меры, например m (количество клозов в формуле) и k (количество выполненных клозов в формуле).

Глава 5

Дальнейшие направления

Другие задачи

Как было сказано выше, подход обобщаем на отличные от L меры. Также было бы интересно изучить автоматизацию доказательств для `SAT` (таблица выигрышей в таком случае становится проще) или `Partial MinSAT`. Но это требует усложнения структуры для хранения формулы.

Улучшение структуры данных

В рамках используемой структуры данных в дереве разбора появляется большое количество дубликатов. Например, в формуле с клозами вида $(xA)(xB)$, будут рассмотрены все возможные размеры подклозов A и B. Хотя достаточно было бы рассматривать только ситуации, когда $|A| \leq |B|$: случай $|A| > |B|$ разбирается аналогично. Разумеется, подразумевается, что ограничения на A и B в контексте одинаковы. В работах по точным алгоритмам это решается с помощью последовательного рассмотрения правил ветвления для формул с фиксированными размерами подклозов (всех или каких-то конкретных).

На данный момент данное улучшение не сделано, так как любое усложнение структуры данных негативно влияет на уверенность в корректности результата.

Эвристики

Используемые эвристики подразумевают исследование наборов формул, получившихся после правил уточнения – по сути алгоритм делает один шаг "вглубь"и выбирает потенциально лучший вариант. Это требует затрат существенного количества времени. Было бы интересно исследовать подход выбора типа уточнения, исходя из анализа таблицы выигрышей

– на практике получается, что чем меньше мажорирующих назначений у формулы, тем проще её упростить. По выигрышам можно оценить, уточнение каких элементов упростит таблицу больше всего. Помимо этого стоит учитывать список покрытий для формулы. Например, если в лучшем покрытии фигурирует подназначение $\{x = 1\}$, то раскрытие соседних с литералом x подклозов потенциально позволит улучшить расщепление для данного покрытия.

Также эвристики плохо работают, если мы заранее не знаем, переменные из какого клоза лучше специфицировать – формула $F = (\overline{x}y^{\theta}C)(xA)(xB)F'$ для `(n,3)-MaxSAT` была выбрана именно из тех соображений, что если рассматривать формулу $F = (xA)(xB)(\overline{x}y^{\theta}C)F'$, облитерация переменной y происходит после раскрытия клозов A и B, что сильно увеличивает дерево разбора.

SMT-солверы

В разделе Проверка корректности программы описывался подход, позволяющий проверить корректность покрытий с помощью SMT-солверов. Обещающим направлением выглядит генерация покрытий с помощью SMT-солверов – теоретически можно перебирать различные варианты покрытий и проверять их корректность, но на практике это будет работать слишком долго.

Верификация

Насколько нам известно, ранее не предпринимались попытки верифицировать корректность доказательства верхних оценок на время работы экспоненциальных алгоритмов. Даже несмотря на то, что в таких доказательствах разборы случаев могут быть достаточно объёмны. В книге [21] для доказательства оценок на время работы алгоритмов использовался следующий подход: для функции f явно определяется функция T_f, такая что $T_f(x)$ моделирует время вычисления $f(x)$.

Нам видится перспективным подход с описанием узлов дерева разбора индуктивным типом `Node` с желаемой константой расщепления c в качестве зависимого параметра, таким, что

- листы описываемы этим типом только если найдётся хорошее расщепление или упрощение, а
- внутренние узлы описываемы только если все элементы какого-либо из уточнений описываются кем-либо из детей узла, а также у них всех такой же параметр c.

Глава 6

Заключение

В процессе работы была построена программа, позволяющая автоматически генерировать экспоненциальные алгоритмы для задачи `MaxSAT`. Как и в работе [12], на вход даётся формула с желаемой константой ветвления и строится дерево доказательство похожей структуры. Отличия нашей программы:

- модель описания формул предполагает наличие необлитированных переменных: если покрытие означивает литералы, соседствующие с какой-то переменной, то для получения хорошего вектора ветвления зачастую неважно, облитерована она или нет. При этом наличие возможности не облитеровывать переменных существенно уменьшает размеры получаемых деревьев доказательств;
- правила упрощения также построены на основе анализа таблицы выигрышей, но упрощаемые формулы подбираются не за счёт подстановок в литералы значений (e.g. $x = 1$) или других литералов (e.g. $x = \overline{y}$), а за счёт перебора возможных формул, имеющих те же мажорирующие выигрыши. Это позволяет чаще находить упрощения, что также сказывается на размере доказательств.

На основе подхода была построена программа, генерирующая точный алгоритм для задачи $(n, 3) - MaxSAT$, параметризированной относительно количества переменных, со временем работы $\mathcal{O}^*(1.175^n)$, что является улучшением алгоритма, предложенного в [5], со временем работы $\mathcal{O}^*(1.191^n)$.

Также был исследован подход, позволяющий проверять корректность правил ветвления с помощью SMT-солверов, что может быть полезно не только для автоматических доказательств.

Литература

[1] Berg Jeremias, Saikko Paul, Järvisalo Matti. Improving the Effectiveness of SAT-Based Preprocessing for MaxSAT // Proceedings of the 24th International Conference on Artificial Intelligence. — IJCAI'15. — AAAI Press, 2015. — P. 239–245. — URL: https://dl.acm.org/doi/10.5555/2832249.2832282.

[2] Zengler Christoph, Küchlin Wolfgang. Boolean Quantifier Elimination for Automotive Configuration - A Case Study // Formal Methods for Industrial Critical Systems. — Formal Methods for Industrial Critical Systems. Springer Berlin Heidelberg, 2013. — P. 48–62. — URL: http://dx.doi.org/10.1007/978-3-642-41010-9_4.

[3] Iterative and Core-Guided Maxsat Solving: a Survey and Assessment / Antonio Morgado, Federico Heras, Mark Liffiton et al. // Constraints. — 2013. — Vol. 18, no. 4. — P. 478–534. — URL: http://dx.doi.org/10.1007/s10601-013-9146-2.

[4] Williams Ryan. A new algorithm for optimal 2-constraint satisfaction and its implications // Theoretical Computer Science. — 2005. — Dec. — Vol. 348, no. 2–3. — P. 357–365. — URL: http://dx.doi.org/10.1016/j.tcs.2005.09.023.

[5] Belova Tatiana, Bliznets Ivan. Algorithms for (n,3)-MAXSAT and parameterization above the all-true assignment // Theoretical Computer Science. — 2020. — Jan. — Vol. 803. — P. 222–233. — URL: http://dx.doi.org/10.1016/j.tcs.2019.11.033.

[6] Goemans Michel X., Williamson David P. New $\frac{3}{4}$-Approximation Algorithms for the Maximum Satisfiability Problem // SIAM Journal on Discrete Mathematics. — 1994. — Nov. — Vol. 7, no. 4. — P. 656–666. — URL: https://doi.org/10.1137/S0895480192243516.

[7] Greedy Algorithms for the Maximum Satisfiability Problem: Simple Algorithms and Inapproximability Bounds / Matthias Poloczek, Georg Schnitger, David P. Williamson, Anke van Zuylen // SIAM Journal on Computing. — 2017. — Jan. — Vol. 46, no. 3. — P. 1029–1061. — URL: http://dx.doi.org/10.1137/15M1053369.

[8] Bansal Nikhil, Raman Venkatesh. Upper Bounds for MaxSat: Further Improved // Algorithms and Computation. — Algorithms and Computation. Springer Berlin Heidelberg, 1999. — P. 247–258. — URL: http://dx.doi.org/10.1007/3-540-46632-0_26.

[9] Resolution and Domination: An Improved Exact MaxSAT Algorithm / Chao Xu, Wenjun Li, Yongjie Yang et al. // Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence. — 2019. — 8. — P. nil. — URL: http://dx.doi.org/10.24963/ijcai.2019/166.

[10] Alferov Vasily, Bliznets Ivan. New Length Dependent Algorithm for Maximum Satisfiability Problem // Proceedings of the AAAI Conference on Artificial Intelligence. — 2021. — May. — Vol. 35, no. 5. — P. 3634–3641. — URL: https://ojs.aaai.org/index.php/AAAI/article/view/16479.

[11] Nikolenko Sergey, Sirotkin Alexander. Worst-case upper bounds for SAT: automated proof. — 2003. — 04.

[12] Fedin Sergey, Kulikov Alexander. Automated Proofs of Upper Bounds on the Running Time of Splitting Algorithms. — 2004. — 10. — P. 248–259. — URL: https://link.springer.com/chapter/10.1007/978-3-540-28639-4_22.

[13] Kulikov Alexander S. Automated Generation of Simplification Rules for SAT and MAXSAT // Theory and Applications of Satisfiability Testing. — Theory and Applications of Satisfiability Testing. Springer Berlin Heidelberg, 2005. — P. 430–436. — URL: http://dx.doi.org/10.1007/11499107_35.

[14] Raman Venkatesh, Ravikumar B., Rao S.Srinivasa. A Simplified Np-Complete Maxsat Problem // Information Processing Letters. — 1998. — Vol. 65, no. 1. — P. 1–6. — URL: http://dx.doi.org/10.1016/s0020-0190(97)00223-8.

[15] Kulikov A. S., Kutskov K. New Upper Bounds for the Problem of Maximal Satisfiability // Discrete Mathematics and Applications. — 2009. —

Vol. 19, no. 2. — P. nil. — URL: http://dx.doi.org/10.1515/dma.2009.009.

[16] Bliznets I. A. A New Upper Bound for (n, 3)-MAX-SAT // Journal of Mathematical Sciences. — 2012. — Vol. 188, no. 1. — P. 1–6. — URL: http://dx.doi.org/10.1007/s10958-012-1101-z.

[17] Chen Jianer, Xu Chao, Wang Jianxin. Dealing With 4-variables By Resolution: an Improved Maxsat Algorithm // Theoretical Computer Science. — 2017. — Vol. 670, no. nil. — P. 33–44. — URL: http://dx.doi.org/10.1016/j.tcs.2017.01.020.

[18] An Improved Branching Algorithm for (n, 3)-MaxSAT Based on Refined Observations / Wenjun Li, Chao Xu, Jianxin Wang, Yongjie Yang // Combinatorial Optimization and Applications. — Combinatorial Optimization and Applications. Springer International Publishing, 2017. — P. 94–108. — URL: http://dx.doi.org/10.1007/978-3-319-71147-8_7.

[19] Kullmann Oliver, Luckhardt Heinz-Dirk. Algorithms for SAT/TAUT decision based on various measures. — 1998.

[20] Fomin Fedor V., Kaski Petteri. Exact Exponential Algorithms // Communications of the ACM. — 2013. — Vol. 56, no. 3. — P. 80–88. — URL: https://doi.org/10.1145/2428556.2428575.

[21] Functional Algorithms, Verified! / Tobias Nipkow, Jasmin Blanchette, Gómez-Londoño Alejandro Eberl, Manuel and et al. — 2021. — URL: https://functional-algorithms-verified.org/.

[22] Fixed-Parameter Tractability of Satisfying Beyond the Number of Variables / Robert Crowston, Gregory Gutin, Mark Jones et al. // Algorithmica. — 2012. — Vol. 68, no. 3. — P. 739–757. — URL: http://dx.doi.org/10.1007/s00453-012-9697-4.

[23] Victor Marek. Introduction to Mathematics of Satisfiability. — 2009. — URL: https://doi.org/10.1201/9781439801741.