ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программный проект на тему:

Оптимизация и распараллеливание алгоритма F4 поиска базиса Грёбнера

Выполнил студент:

группы #БПМИ205, 4 курса Стёпкин Степан Максимович

Принял руководитель ВКР:

Колесниченко Елена Юрьевна
Доцент
Факультет компьютерных наук НИУ ВШЭ

Соруководитель:

Трушин Дмитрий Витальевич
Доцент
Факультет компьютерных наук НИУ ВШЭ

Москва 2024

Содержание

Аннотация

Базис Грёбнера - одно из важнейших понятий вычислительной алгебры. Для любой системы полиномиальных уравнений базис Гребнера может определить, имеет ли она решения и бесконечно ли это число. Существует несколько методов нахождения базиса Гребнера. Одним из самых быстрых является алгоритм F4. Скорость важна для любой алгебраической системы, поэтому авторы стараются реализовать алгоритмы с минимальной задержкой, насколько это возможно. В этой статье мы подробно обсудим реализацию F4. Кроме того, мы реализуем эти алгоритмы в библиотеке с открытым исходным кодом и сравним производительность с другими реализациями.

Annotation

The Gröbner basis is one of the most important concepts in computational algebra. For any system of polynomial equations, Gröbner bases can tell whether it has solutions and whether their number is infinite. There are several methods for finding the Gröbner basis. One of the fastest is the F4 algorithm. Speed is important for any algebraic system, so authors try to implement algorithms with as minimal latency as possible. In this paper, we discuss implementation of F4 in details. Futhermore, we implement these algorithms in an open source library.

Ключевые слова

Базис Грёбнера, алгоритм F4, C++, оптимизация программного обеспечения

1 Введение

Алгоритмы по поиску базисов Грёбнера находятся на стыке линейной алгебры и программирования. За всё время существования базисов Грёбнера было придумано несколько алгоритмов и множество оптимизаций по их вычислению.

В данной работе мы реализуем алгоритм F4 со всеми возможными идейными оптимизациями, попытаемся оптимизировать его архитектурно и распараллелим его. При этом мы хотим позволить пользователю максимальную свободу - самому выбирать поле и порядок мономов. Подробно разберем как устроены алгоритмы и какие их части стоит оптимизировать. Весь код и бенчмарки можно найти на GitHub[1]. Библиотека называется FF4, и далее по работе будет использоваться именно это название.

Работа написана в следующем порядке — сначала идут основные определения, затем подробно обсуждается алгоритм Бухбергера, его оптимизации, имплементации и бенчмарки. После этого идет аналогичный анализ F4 — подробно объясняется алгоритм, приводятся все его оптимизации, обсуждается эффективность каждой из них. Затем подробно рассказывается о удачных и неудачных попытках оптимизровать алгоритм, о подробностях имплементации, о распараллеливании и приводится множество бенчмарков. Кроме того, нас ожидает достаточно неожиданный вывод.

Алгоритм реализован как библиотека на языке C++, протестирован на macOS и linux. Все бенчмарки были произведены на MacBook Pro 23, Apple M2 Pro, 32Gb.

2 Базисы Грёбнера и алгоритм Бухбергера

Считается, что математика зародилась вместе с человеком. Тем не менее, даже в наше время продолжают совершаться важные и в то же время доступные широкой публике открытия. Одним из таких примеров является базис Грёбнера, который был представлен вместе с алгоритмом Бухбергера Бруно Бухбергером в рамках его докторской диссертации в 1965 году [1].

2.1 Основные определения

Для базисов Грёбнера необходимо понятие порядка на одночленах и старшего члена. Для многочленов от одной переменной понятие старшего члена изучается в школе - например в $x^3+13x^2-21x+6$ старшим членом является x^3. Тем не менее для многочленов от нескольких

[1] https://github.com/KaurkerDevourer/ff4

переменных ответ уже не является очевидным. Что является старшим членом у многочлена $x_0^3 + x_0x_1^2 + x_2^4$? Для ответа на этот вопрос были придуманы порядки на мономах. Самым базовым из них является **лексикографический**. Перед тем как мы перейдем к определению порядков, поясним, что одночлен x_0^3 представляет собой слово [3], $x_1^2x_3^4$ слово [0204].

Определение 1. ***Лексикографическим порядком****, называют порядок в котором слово A меньше слова B, если*

- *слово A является префиксом слова B.*

  или

- *первые k символов слов совпадают, но $k+1$-символ слова A меньше $k+1$-символа слова B.*

  Например, для лексикографического порядка верно
  $x_0^3 > x_0^2x_1 > x_0^2x_2 > x_1^2x_2 > x_2^3 > x_2$.

У лексикографического порядка есть большой минус — он не смотрит на суммарные степени одночленов, например $x_0 > x_1^{100}x_2^{200}$. Таким образом, смотря на старший член в лексикографическом порядке, сложно выявить какие-то свойства многочлена. Поэтому был придуман порядок, который в данный момент применяется во всех вычислениях базисов Грёбнера (и не только). Академически он называется **степенно обратным лексикографическим порядком**, но во всех работах его называют просто **grevlex** (от англ. Graded reverse lexicographic order).

Определение 2. ***grevlex*** *порядком, называют порядок в котором слово A меньше слова B, если*

- *сумма степеней всех переменных в одночлене A меньше суммы степеней всех переменных в одночлене B.*

  или

- *последние k символов слова совпадают, но $k+1$ символ слова A с конца больше $k+1$-го символа с конца слова B.*

  Например, в порядке *grevlex* верно
  $x_3^3 > x_2^2 > x_0$, так как суммарные степени мономов соответственно равны 3,2,1.

Если же суммарные степени равны, то *grevlex* считает меньшим то слово, в котором обратный лексикографический порядок больше. То есть

$x_0^2 > x_0x_1 > x_1^2 > x_0x_2 > x_1x_2 > x_2^2$.

$x_1^2 > x_0x_2$, так как суммарные степени равны 2, а слова [020] и [101] сравниваются лексикографически в обратном порядке. Так как второе слово больше, то *grevlex* объявляет, что и стоит оно позже.

Таким образом, суммарные степени всех мономов в полиноме будут не больше чем суммарная степень старшего члена. *grevlex* является основным порядком для измерения скорости работы алгоритмов по вычислению базиса Грёбнера.

Определение 3. *Пусть $(f_1, \ldots, f_n)$ множество полиномов. Они называются* ***базисом Грёбнера*** *идеала $I = (f_1, \ldots, f_n)$ если для каждого полинома f из I существует полином f_i из множества $(f_1, \ldots, f_n)$ такой, что старший член f делится на старший член f_i.*

Определение 4. *Базис Грёбнера называется* ***минимальным****, если никакой старший член полинома не делится на старший член любого другого полинома.*

Например (x_0, x_1, x_2) и (x_0x_1, x_0, x_1, x_2) являются базисами Грёбнера для идеала $I = (x_0, x_1, x_2)$, но второй базис не является минимальным.

Несмотря на неочевидные свойства базис Грёбнера невероятно ценен, благодаря двум теоремам

Теорема 1. *Система полиномиальных уравнений не имеет решений тогда и только тогда, когда в базисе Грёбнера присутствует ненулевая константа.*

Теорема 2. *Количество комплексных решений для системы уравнений конечно тогда и только тогда, когда для каждой переменной x_i в базисе Грёбнера присутствует полином, со старшим членом вида x_i^k.*

Таким образом базис Грёбнера отвечает на важнейшие вопросы - есть ли у системы полиномиальных уравнений решения и конечно ли их количество.

2.2 Алгоритм Бухбергера

Первый алгоритм по вычислению базиса Грёбнера был предложен в той же работе, что и сам базис, уже упомянутым Бруно Бухбергером. Однако, прежде чем перейти к алгоритму, приведем основные обозначения и ключевую теорему, на которой основывается алгоритм.

Моном без коэффициента далее будем называть *термом*.

$lm(f)$ - leading monomial, то есть старший член многочлена f.

$lt(f)$ - leading term, то есть старший терм полинома f.

$lot(f) = f - lm(F)$ - то есть полином f без старшего члена.

$mon(f)$ - множество всех термов многочлена f.

lcm - least common multiple, наименьшее общее кратное термов.

gcd - greatest common divisor, наибольший общий множитель термов.

$S \xrightarrow{F} d$ - результатом редукции F над S является d.

Определение 5. ***S-многочленом*** $S(f_i, f_j)$, *называется многочлен*

$S(f_i, f_j) = \frac{L}{lt(f_i)} f_i - \frac{L}{lt(a_j)} f_j$, *где* $L = LCM(lt(f_i), lt(f_j))$.

S-многочлен используется для того, чтобы получить полином с меньшим старшим членом в том же идеале. Например

$S(xy - z, x^2 - y) = x * (xy - z) - y(x^2 - y) = x^2y - xz - x^2y + y^2 = -xz + y^2$.

То есть $lt = xz$, хотя до этого был xy и x^2. Напомним, что при любых двух упомянутых выше порядках $x^2 > xy > xz$.

Теорема 3. *(Критерий Бухбергера) Пусть* $F = \{f_1, \ldots, f_k\} \subset I$ *для некоторого идеала* I. *Если* $S(f_i, f_j)$ *редуцируются к нулю относительно* F *для всех пар* (i, j), *то* F *является базисом Грёбнера для* I.

Таким образом мы уже умеем проверять для данного набора полиномов, является ли он базисом Грёбнера своего идеала - перебираем все пары (i, j) и проверяем, редуцируются ли S-многочлены к нулю.

Заметим, что редуцированный S-многочлен лежит в идеале. Почему бы его не добавить в текущую систему полиномов и продолжить перебор пар? Ровно на такой идее и основан первый алгоритм по вычислению базисов Грёбнера - алгоритм Бухбергера.

Algorithm 1: Buchberger algorithm

Input : Set of polynomials F for ideal I

Output: Groebner basis F for ideal I

$P = \{S(F_i, F_j) \mid 0 \leq i < j < size(F)\}$

while $P.size() > 0$ **do**

  $S_{ij} = first(P)$

  $P = P \setminus \{S_{ij}\}$

  $h = Reduce(S_{ij}, G)$

  if $h \neq 0$ **then**

    $F = F \cup h$

    $P = P \cup \{S(F_i, h) \mid 0 \leq i < size(F)\}$

  end

end

На вход алгоритм Бухбергера получается множество полиномов, после чего добавляет элементы в множество, пока оно не станет базисом Грёбнера. Для начала ответим на самый базовый вопрос - почему алгоритм завершится. Исторически и формально это вытекает из леммы Диксона. Говоря простым языком - S-полином сокращает старшие члены любых двух многочленов. Чем дальше идет алгоритм тем меньшие старшие члены появляются в базисе. В конце концов - либо все $S \xrightarrow{F} 0$, либо в базис добавится ненулевая константа. А константа редуцирует до нуля всё в своём идеале.

3 Оптимизации алгоритма Бухбергера

Так как алгоритм Бухбергера появился одновременно с базисами Грёбнера, несложно поверить, что он работает не оптимально. В этой работе не будет обсуждаться O-сложность алгоритмов. Связано это в основном с тем, что оценка O-сложности приведенных алгоритмов слишком тяжеловесна для работы такого объема и уровня. Если читателю интересно ознакомиться с подобными работами, то вот пару примеров: [2, 3]. В этой работе, как и в большинстве работ по базисам Грёбнера упор будет делаться на оптимизацию времени выполнения алгоритмов.

Несмотря на то, что работа посвящена F4, на изучению Бухбергера и его оптимизаций было потрачено много времени, так как это основа для эффективного F4.

3.1 Основные идеи

Одной из идей является оптимизация выбора пары. Утверждается, что выбирая пару с минимальным старшим членом среди всех S-пар алгоритм завершится быстрее. Это важно и дает улучшения в производительности, но тривиально. Гораздо интереснее оптимизировать S-редукции.

С самого существования алгоритмов по вычислению базисов Грёбнера и до сих пор, большой упор ведется в предсказание того, что S-полином будет отредуцирован к нулю. Связано это с тем, что большинство S-полиномов (90%) будут отредуцированы к нулю, и как следствие, проведя время вычислений алгоритм не изменит свой базис. Поэтому стараются находить всё большие способы заранее понимать, что определенный S-полином будет отредуцирован к нулю, тем самым сокращая время работы алгоритма.

Существуют два основных критерия, по которым можно быстро понять, что вычисления для определенного S-полинома не нужны, так как он будет отредуцирован к нулю. Во

многих источниках они называются и определяются по-разному, являясь по-сути одним и тем же. Мы сохраним первоначальную терминологию.

Критерий 1. *НОД критерий Бухбергера (Buchberger's gcd criterion).*

Если p_1 и p_2 полиномы у которых $gcd(lt(p_1), lt(p_2)) = 1$, то $S(p_1, p_2) \xrightarrow{\{p_1,p_2\}} 0$.

Доказательство. Пусть p_1 и p_2 полиномы с взаимно простыми старшими членами. Тогда $S(p_1, p_2) = lt(p_2)p_1 - lt(p_1)p_2 = lt(p_2)lot(p_1) - lt(p_1)lot(p_2) \xrightarrow{p_2} lt(p_2)lot(p_1) - p_2lot(p_1) - lt(p_1)lot(p_2) = -lot(p_2)lot(p_1) - lt(p_1)lot(p_2) = -p_1lot(p_2) \xrightarrow{p_1} 0$ □

Критерий 2. *НОК критерий Бухбергера (Buchberger's lcm criterion).*

Если p_1, p_2 и p_3 полиномы в F, при этом $S(p_1, p_2) \xrightarrow{F} 0$ и $S(p_2, p_3) \xrightarrow{F} 0$ и $lt(p_2) \mid lcm(lt(p_1), lt(p_3))$ то $S(p_1, p_3) \xrightarrow{F} 0$.

Доказательство. Пусть p_1, p_2 и p_3 вышеупомянутые полиномы.

$$k = lcm(lt(p_1), lt(p_3)), k_1 = lcm(lt(p_1), lt(p_2)), k_2 = lcm(lt(p_3), lt(p_2)).$$

$lcm(lcm(a, b), lcm(b, c)) = lcm(a, b, c)$, но если b делит $lcm(a, c)$, то $lcm(a, b, c) = lcm(a, c)$.

$$S(p_1, p_2) = \frac{k_1}{lt(p_1)}p_1 - \frac{k_1}{lt(p_2)}p_2 \xrightarrow{F} 0$$

$$S(p_2, p_3) = \frac{k_2}{lt(p_2)}p_2 - \frac{k_2}{lt(p_3)}p_3 \xrightarrow{F} 0$$

$$\frac{lcm(k_1,k_2)}{k_2}S(p_2, p_3) + \frac{lcm(k_2,k_1)}{k_1}S(p_1, p_2) = \frac{lcm(k_1,k_2)}{lt(p_1)}p_1 - \frac{lcm(k_1,k_2)}{lt(p_3)}p_3 =$$

$$= \frac{k}{lcm(k_1,k_2)}S(p_1, p_3) = S(p_1, p_3) \xrightarrow{F} 0$$ □

3.2 GMI [4]

На основе этих двух критериев был придуман алгоритм обновления пар (i, j), которые нужно проверять на редуцируемость и позже возможно добавлять в базис.

Algorithm 2: Gebauer-Möller installation (GMI)

Input : Current set of polynomials F, new element h to add into set of polynomials, current set of pairs to check P

Output: Updated set of polynomials F, updated set of pairs to check P

$P' = \{S(F_i, h) \mid 0 \leq i < size(F)\}$

for $(f, g) \in P$ **do**
  $L = lcm(lt(f), lt(g))$
  if $lt(h) \mid L$ ***and*** $lcm(lt(h), lt(f)) \neq L$ ***and*** $lcm(lt(h), lt(g)) \neq L$ **then**
    $P = P \setminus \{(f, g)\}$
  end
end
for $(f, h) \in P'$ **do**
  for $(g, h) \in P' \setminus \{(f, h)\}$ **do**
    if $lcm(lt(f), lt(h)) \mid lcm(lt(g), lt(h))$ **then**
      $P' = P' \setminus \{(g, h)\}$
    end
  end
end
for $(f, h) \in P'$ **do**
  if $gcd(lt(f), lt(h)) = 1$ **then**
    $P' = P' \setminus \{(f, h)\}$
  end
end
$F \cup \{h\}$
$P \cup \{P'\}$

GMI позволяет не производить вычисления для некоторых S-полиномов, предсказывая их редуцируемость к нулю. Первые два цикла — lcm критерий, последний — gcd критерий. Корректность таких действий мы доказали выше. Тогда модифицированный алгоритм Бухбергера, не делающий часть нулевых редукций, и выбирающий S-полином с минимальным старшим членом выглядит следующим образом:

Algorithm 3: Improved buchberger algorithm

Input : Set of polynomials F for ideal I

Output: Groebner basis F for ideal I

$P = \{\}$

for $h \in F$ **do**
  $GMI(F, h, P)$

end

while $P.size() > 0$ **do**
  $S_{ij} = Select_{min}(P)$
  $P = P \setminus \{S_{ij}\}$
  $h = Reduce(S_{ij}, G)$
  if $h \neq 0$ **then**
    $GMI(F, h, P)$
  end

end

4 Реализация алгоритма Бухбергера

В этой главе мы поговорим о деталях имплементации алгоритма Бухбергера — какие части есть и что они значат. Вся имплементация произведена на C++ и находится в репозитории. Реализованы поля **Rational** и **PrimeModular**, порядки **GrevLex** и **Lexicographical** - на деталях их реализации мы останавливаться не будем.

4.1 Term

```
class Term {
    ...
    private:
        std::vector<uint16_t> data;
}
```

Класс одночленов без коэффициентов. **data[i]** хранит степень для x_i переменной в терме. Выделяется в отдельный класс он по многим причинам, основной из которых является корректность операций. Так, например **lcm** и **gcd** определены для **Term**, но не для мономов. Главное за чем следит **Term** - корректность состояние **data**. В **data** не допускаются нули на конце, кроме случая **data.size() == 1**. **data = [0]** соответствует константе. Нужно это для экономии памяти. Приведем примеры для большего понимания:

$x_0x_1x_2$ будет лежать в $data$ как $[1, 1, 1]$.

x_2^3 будет лежать в $data$ как $[0, 0, 2]$.

$l = lcm(x_0x_1x_2, x_2^3) = lcm([1, 1, 1], [0, 0, 2]) = [1, 1, 2]$.

$l/x_2^3 = [1, 1, 2] - [0, 0, 2] = [1, 1, 0] = [1, 1]$.

$[1, 1]/[1] = [0, 1]$. $[1, 1] * [1] = [1, 2]$. $gcd([1, 2], [0, 3, 1]) = [0, 2, 0] = [0, 2]$

4.2 Monomial

```
template <typename TCoef>
class Monomial {
    ...
    private:
        Term term;
        TCoef coef;
}
```

Класс одночлена с коэффициентом. Класс шаблонизирован для разных полей **TComp**. Напомним, что в библиотеке присутствуют основные для базисов Грёбнера типы **Rational** и **PrimeModular**.

4.3 Polynomial

```
template <typename TCoef, typename TComp>
class Polynomial {
    ...
    private:
        std::vector<Monomial<TCoef>> monomials;
}
```

Класс многочлена. Шаблонизирован для полей и для разных порядков. Мономы в векторе лежат строго в **TComp** порядке. Это позволяет работать с многочленами за **O(monomials.size())**. Например, при сложении/вычитании двух полиномов мы можем использовать метод двух указателей:

```
size_t i = 0, j = 0;
std::vector<Monomial<TCoef>> newMonomial;
while (i < left.size() && j < right.size()) {
    if (left[i] < right[j]) {
        newMonomial.push_back(left[i]);
        i++;
    } else if (right[j] < left[i]) {
        newMonomial.push_back(right[j]);
        j++;
    } else {
        newMonomial.push_back(left[i] + right[j]);
        i++;
        j++;
    }
}
...
```

Отметим, что множеством полиномов в стандартном алгоритме Бухбергера это **std::vector**, а в улучшенном Бухбергере с F4 **std::set**, так как есть функция **EraseLead** и итерация по всем полиномам линейна в обоих структурах.

4.4 EraseByLead

Algorithm 4: EraseByLead

Input : Set of polynomials F for ideal I, new element h to add into set of polynomials

Output: Reduced set of polynomials F for ideal I

for $g \in F$ **do**

  if $lt(g) \mid lt(h)$ **then**

    $F = F \setminus \{g\}$

  end

end

EraseByLead - функция, которой мы дополним **GMI**. Мы не обсуждали эту функцию выше, так как на производительность она почти не влияет. Её смысл в том, что прежде чем добавить полином h в рассматриваемый базис, мы удаляем из него полиномы, старший член которых делится на старший член h, в связи с тем что они нам больше никогда не понадобятся. Это позволяет улучшенному алгоритму Бухбергера выдавать в ответе минимальный базис Грёбнера (чем не может похвастаться базовый алгоритм Бухбергера). К тому

же понятно, что чем меньше множество полиномов которое мы обрабатываем, тем быстрее код.

4.5 Benchmark

Несмотря на простоту подхода, во многих библиотеках одночлены хранятся в **std::list** или **std::set**. Кроме лучшей асимптотики, подход с **std::vector** хранит одночлены в векторе эффективно по памяти - лишний раз не проиграем по кэшу и будем всегда точно знать сколько памяти используем. Также подобное хорошо сочитается с моделью памяти **Term**. Сравним это с реализацеий на **std::set** GroebnerBasisLib[2].

Тест	ff4::Buchberger	GroebnerBasisLib::Buchberger	ff4::ImprovedBuchberger
cyclic4	1.04 мс	2.42 мс	2.15 мкс
sym3-3	217 мкс	2.85 мс	3.65 мкс
katsura4	21.95 мс	20.98 мс	58 мкс
katsura5	139.62 мс	392.34 мс	4.7 мс

Таблица 4.1: Измерения производительности алгоритмов Бухбергера для $p = 31$, grevlex

ff4::Buchberger кратно лучше GroebnerBasisLib::Buchberger на многих тестах (за счет модели памяти), в то время как ff4::ImprovedBuchberger за счет **GMI** для них просто недостижим. Далее в работе будем сравнивать именно ff4::ImprovedBuchberger с F4.

5 Алгоритм F4

Алгоритм F4 был опубликован Jean-Charles Faugère в 1999 [5]. Отличиет алгоритм F4 от алгоритма Бухбергера - функции **Select** и **Reduce**. Если есть желание ознакомиться с оригинальной работой, для начала порекомендую ознакомиться с её кратким описанием [6], после чего чтение изначальной работы упростится кратно.

Для начала опишем все методы, а затем подробно объясним как работает алгоритм. Во-первых введем новый объект - **CriticalPair**.

[2] https://github.com/cdraugr/GroebnerBasis

5.1 CriticalPair

По сути, это будет всё та же пара (i, j), только мы сразу подсчитаем **TotalDegree**. $TotalDegree(term) = \sum_{i=0}^{size(term)} term[i]$. Далее будем обозначать это как $deg(term)$.

```
class CriticalPair {
    ...
    private:
        Polynomial left; // Left polynomial
        Polynomial right; // Right polynomial
        Term lcm; // lcm(lt(left), lt(right))
        Term::Degree degree; // deg(lcm)
}
```

5.2 Select

Отметим, что сам алгоритм F4 позволяет использовать любую функцию **Select**. Тем не менее наиболее эффективной считается следующая:

Algorithm 5: Select

Input : Current set of critical pairs P

Output: Set of critical pairs to check on current step P', Updated set of critical pairs P

$P' = \{\}$

$d = min_{def}(P)$

for $cp \in P$ **do**

  if $def(cp) == d$ **then**

    $P = P \cup \{cp\}$

  end

end

$P = P \setminus P'$

Таким образом мы выбираем все критические пары, с минимальной суммарной степенью. Отличие от улучшенного Бухбергера - выбираются сразу все такие пары, а не одна. Более того, если выбирать ровно 1 такую пару, то алгоритм сводится к улучшенному алгоритму Бухбергера.

5.3 Reduce

Algorithm 6: Reduce

Input : Set of critical pairs P', set of polynomials F

Output: New polynomials to add into the basis G

$L = SymbolicPreprocessing(P', F)$

$M =$ matrix with rows as polynomials in L

$M' = Reduce(M)$

$L' =$ Polynomials(M')

$G = \{g \mid LT(g) \neq LT(F) \forall f \in F\}$

Если раньше мы просто редуцировали один полином другими, то теперь мы сделали матрицу M, где строки это полиномы, столбцы это термы отсортированные в заданном порядке начиная с наибольшего, а значения в матрице - коэффициент терма в полиноме. И точно так же редуцируем (приводим к треугольному виду) только теперь матрицу. Но что такое **SymbolicPreprocessing**?

5.4 SymbolicPreprocessing

Algorithm 7: Symbolic Preprocessing

Input : Set of critical pairs P', set of polynomials F

Output: Set of polynomials L

$L = \{\}$

for $p \in P'$ **do**

  $L = L \cup \{\frac{p.lcm}{lt(p.left)} \cdot p.left\}$

  $L = L \cup \{\frac{p.lcm}{lt(p.right)} \cdot p.right\}$

end

$done = \{lt(l) | l \in L\}$

while $done \neq mon(L)$ **do**

  $t = max_{term}(L \setminus done)$

  $done = done \cup \{t\}$

  for $f \in F$ **do**

    if $t \mid lt(f)$ **then**

      $L = L \cup \{\frac{t}{lt(f)} \cdot f\}$

      break

    end

  end

end

Так определяется метод **Symbolic Preprocessing**. Потом L подаётся на вход в матрицу. Первый цикл, по сути добавляет S-пару в матрицу — если вычесть из верхнего нижнее:

$\frac{p.lcm}{lt(p.left)} \cdot p.left \ - \frac{p.lcm}{lt(p.right)} \cdot p.right = S(left, right)$

Второй цикл добавляет в матрицу все возможные редукторы этих S-пар. Так, получается, что за одну редукцию матрицы мы редуцируем сразу множество S-пар.

6 Оптимизация алгоритма F4

Несмотря на то, что стандартный алгоритм F4 относительно Бухбергера работает на порядки быстрее, некоторые идеи позволяют ускорить его кратно. Во-первых, всё тот же **GMI**. Он абсолютно идентичен, поэтому мы его не обсуждаем. Самое интересное - оптимизация матричных вычислений

6.1 GBLA[7]

Для начала вспомним, как строятся матрицы - сначала добавляются S-пары, а потом их редукторы.

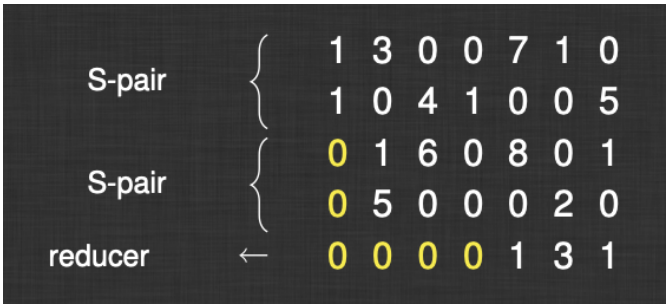

Рис. 6.1: Внутренняя структура матрицы F4 [8]

Заметим, что для такой матрицы очень легко найти pivot row and columns. А давайте их переставим!

Рис. 6.2: Переставленные ряды и строки матрицы [8]

Теперь, скажем что матрица выглядит следующим образом: $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$. Тогда вместо просто редукции матрицы, сделаем следующие операции:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \xrightarrow{TRSM} \begin{pmatrix} I & A^{-1}B \\ C & D \end{pmatrix} \xrightarrow{AXPY} \begin{pmatrix} I & A^{-1}B \\ 0 & D - CA^{-1}B \end{pmatrix} \xrightarrow{reduce(D)} \begin{pmatrix} I & A^{-1}B \\ 0 & red(D - CA^{-1}B) \end{pmatrix}.$$

Вся эта красота нам нужна для скорости. А сильно быстрее мы, потому что для любой матрицы в F4 верно (N - количество строк, M - количество столбцов, piv - количество строк квадратной матрицы A):

$piv >> N - piv$ и $piv >> M - piv$

Таким образом вместо первоначальной $O(N^3)$, мы получаем $O(piv^2 * (M - piv)) + O(piv(M - piv)(N - piv)) + O((N - piv)^3) \approx O(piv^2)$. То есть кроме того, что мы уменьшили степень, мы еще и уменьшили переменную от которой зависит оценка (хоть и не сильно).

В частности, такая оптимизация дала ускорение в 4 раза на **katsura-9**.

6.2 Simplify

В работах по алгоритму F4 и в его имплементациях на каждом шаге встречается метод **Simplify**. Он запоминает все изначальные полиномы которые вошли в матрицу, а затем результат редукции. Далее он пытается переиспользовать эти результаты, чтобы еще быстрее редуцировать полиномы. В некоторых работах даже утверждается, что F4 такой быстрый, именно благодаря данному методу. Тем не менее мои результаты сильно расходятся с этой теорией.

Реализация алгоритма была произведена на F4 с GBLA и сильно замедлила алгоритм. Связано это с тем, что очень дорого хранить результаты всех вычислений. Далее, будет понятно, почему это не проблема и почему нормально не использовать **Simplify**.

6.3 Extended Criteria

В 2007 году была опубликована работа "An extension of Buchberger's criteria for Gröbner basis decision"[9, 10, 11]. На протяжении 18 страниц, там доказывается новый **Extended Criteria**, который включает в себя **gcd** и **lcm** критерии Бухбергера. Приведем сам критерий:

Критерий 3. *(Extended criteria). Пусть p_1, p_2 и p_3 полиномы в F, и $t_1 = lt(p_1), t_2 = lt(p_2), t_3 = lt(p_3)$. Мы говорим что они соответствуют* ***расширенному критерию*** *если выполнены оба условия:*

- $t2 \mid gcd(t_1, t_3)$ *или* $lcm(t_1, t_3) \mid t_2$.
- *Для всех переменных x_i выполнено одно из двух условий:*
  - $min(t_1[i], t_3[i]) = 0$
  - $t_2[i] \leq max(t_1[i], t_3[i]$

Такой критерий легко добавить в **GMI**, рядом с **LCM** критерием.

Тут стоит чуть подробнее написать про критерии Бухбергера как сущность в линейной алгебре и в программировании. Если обратиться к источникам, то там можно найти, что **lcm** критерий и **расширенный критерий** определены для некоторой цепочки $(t_1, \cdots, t_k)$. Что это за цепочки и почему они не встречаются ни в одной из реализаций алгоритмов?

Цепочки можно представлять как путь, в котором ребрами связаны полиномы, которые отредуцировались к нулю. Во всех реализациях же, цепочки содержат максимум 3 полинома. Связано это на самом деле с простым фактом - **любой** полином в алгоритмах по вычислению базиса Грёбнера когда-то отредуцируется к нулю. Действительно, если какой-то S-полином не отредуцировался к нулю, то мы его добавили в базис. А значит, уже с новым базисом он редуцируется к нулю. Поэтому не имеет смысла поддерживать никакие цепочки, достаточно лишь смотреть на пары $(p_1, p_2), (p_2, p_3)$.

Именно благодаря такому свойству, есть разница между теорией и имплементацией в критериях. Утверждать, что любой полином отредуцируется к нулю в текущем базисе нельзя, потому в теоретических работах возникают цепочки.

Имлементирован **Extended criteria** был последним, и стал решающим фактором, благодаря которому ff4 догнала openf4 по перфомансу. Если говорить о цифрах, то алгоритм стал работать на 30% быстрее. Объясняется это просто — если оптимизации матричных вычислений позволяют быстрее производить то или иное действие на конкретной матрице, то критерии позволяют исключить множество полиномов из этих матриц в самом начале.

6.4 NOTGBLA

Несмотря на то, что такой подход описан в работе **GBLA**, по какой-то причине на него не обращено должного внимания. Даже названия для него не выделили. Так что назовем его **NOTGBLA**. Напомним результат **GBLA**: $\begin{pmatrix} A & B \\ C & D \end{pmatrix} \xrightarrow{GBLA} \begin{pmatrix} I & A^{-1}B \\ 0 & red(D - CA^{-1}B) \end{pmatrix}$

Вспомним что вообще делает алгоритм матричной редукции — какие полиномы являются результатом редукции? Только полиномы из правой нижней части. Тогда **NOTGBLA** предлагает делать следующее:

$$\begin{pmatrix} AB \\ CD \end{pmatrix} \xrightarrow{reduce(CD)\ by\ (AB)} \begin{pmatrix} AB \\ CD - C \cdot AB \end{pmatrix} \xrightarrow{reduce(D')\ by\ itself} \begin{pmatrix} A & B \\ 0 & reduce(D') \end{pmatrix}$$

Если посмотреть на ассимптотику, то получим $O(piv * (N - piv) * M) + O((N - piv)^3) \approx O(piv * M)$. Выглядит больше чем было.

Так и есть. Первая попытка была неудачной и матричная редукция сильно замедлилась.

В этот момент в игру вступает разряженность матриц. У изначальной входной матрицы, примерно 95% значений равны нулю. Тогда, давайте для каждой строчки из (A B) заранее посчитаем где нули, а где нет. И будем обновлять в (C D) используя только ненулевые значения из (A B). Заметим, что в данном подходе (A B) не изменяется, а значит мы сможем использовать предподсчет в течении всей редукции.

Подобный подсчет ненулевых коэффициентов в **GBLA** не работает эффективно, так как значения постоянно везде изменяются, а там где не изменяются, таких значений в любом случае очень мало $N - piv$ или же $M - piv$.

В итоге, **NOGBLA** обгоняет **GBLA** подход на 25%.

Заметим, что в данном подходе, **Simplify** является абсолютно неэффективным, так как большая часть матрицы не изменяется.

7 Реализация алгоритма F4

Так как алгоритм большой, то опишем только интересные и неочевидные части. С полной версией можно ознакомиться на GitHub[3].

Реализации порядков, полей и внутренних классов уже обсудили выше, в части про реализацию алгоритма Бухбергера.

[3] https://github.com/KaurkerDevourer/ff4

7.1 UpdateL

Функция используемая для нахождения редукторов в **SymbolicPreprocessing**

```
void UpdateL(TPolynomials& L,
             Term& term,
             TPolynomialSet& polynomials,
             TTermHashSet& diff,
             TTermHashSet& done) {
    for (const auto& polynomial : polynomials) {
        const auto& t = polynomial.GetLeadingTerm();
        if (term.IsDivisibleBy(t)) {
            Polynomial reducer = (term / t) * polynomial;
            L.push_back(std::move(reducer));
            for (const auto& m : L.back().GetMonomials()) {
                if (!done.contains(m.GetTerm())) {
                    diff.insert(m.GetTerm());
                }
            }
            break;
        }
    }
}
```

done - уже обработанные термы, **diff** - термы которые еще нужно обработать.

7.2 Matrix

Matrix реализована на стандартном векторе, обращение к элементам идет через operator(). Были попытки хранить A, B, C, D отдельно, или немного внутренне переставить индексацию, но самым быстрым вариантом оказалось ничего подобного не делать.

7.3 FillMatrix

Эта функция интересна тем, что мы сразу заполняем матрицу в формате **GBLA**, вместо того, чтобы потом переставлять ряды и столбцы.

```
template <typename TCoef, typename TComp>
size_t FillMatrix(TPolynomials<TCoef, TComp>& F,
```

```
            Matrix<TCoef>& matrix,
            std::vector<Term>& vTerms,
            const std::vector<Term>& diffSet,
            std::vector<std::vector<size_t> >& nnext) {
size_t cnt = 0;
size_t swp = 0;
std::vector<bool> not_pivot(F.size());
TTermHashSet leadingTerms;
    // storing leading terms, to find pivots
std::unordered_map<Term, size_t> Mp;
    // mapping term -> column
for (size_t i = 0; i < F.size(); i++) {
    auto [_, inserted] =
        leadingTerms.insert(F[i].GetLeadingTerm());
    if (!inserted) {
        not_pivot[i] = true;
        swp++;
        continue;
    }
    Mp[F[i].GetLeadingTerm()] = cnt;
    vTerms[cnt] = F[i].GetLeadingTerm();
    cnt++;
}

cnt = diffSet.size() - 1;
for (auto& term : diffSet) {
    if (Mp.find(term) == Mp.end()) {
        Mp[term] = cnt;
        vTerms[cnt] = term;
        cnt--;
    }
}

nnext.reserve(F.size() - swp);
for (size_t i = 0, j = 0; i < F.size(); i++) {
    if (not_pivot[i]) {
```

```
                j++;
                continue;
            }
            std::vector<size_t> next;
            next.reserve(F[i].GetMonomials().size());
            for (const auto& m : F[i].GetMonomials()) {
                const auto& term = m.GetTerm();
                size_t column = Mp[term];
                matrix(i - j, column) = m.GetCoef();
                next.push_back(column);
            }
            nnext.push_back(std::move(next));
            }
        }

        for (size_t i = 0, j = 0; i < F.size(); i++) {
            if (!not_pivot[i]) {
                continue;
            }
            for (const auto& m : F[i].GetMonomials()) {
                const auto& term = m.GetTerm();
                matrix(F.size() - 1 - j, Mp[term]) = m.GetCoef();
            }
            j++;
        }

        return F.size() - swp;
    }
```

7.4 NOTGBLA

Приведем метод редукции нижних рядов верхними **NOTGBLA**, из интересного - **std::vector<std::vector<size_t» nnext**, тот самый предподсчитанный вектор который делает этот алгоритм быстрее **GBLA**.

```
template <typename TCoef>
void NOTRSM(Matrix<TCoef>& matrix,
```

```
                size_t pivots,
                const std::vector<std::vector<size_t> >& nnext) {
        for (size_t i = 0; i < pivots; i++) {
            const auto& next = nnext[i];
            for (size_t j = pivots; j < matrix.N_; j++) {
                if (matrix(j, i) != 0) {
                    TCoef factor = matrix(j, i);
                    for (size_t k = 0; k < next.size(); k++) {
                        matrix(j, next[k]) -= factor * matrix(i, next[k]);
                    }
                }
            }
        }
    }
```

7.5 Распараллеливание

Неожиданным стало, что распараллеливать в полученном алгоритме в общем то нечего. То есть, конечно, можно распараллелить **NOTRSM**, а именно часть с вычитанием полинома, но, оказывается, что так как это очень маленькая часть, то threading только замедляет. Ознакомиться можно в пулл-реквесте GitHub[4]. В матричных вычислениях кроме **NOTRSM** нечего распараллеливать, как и во всем остальном коде. Поэтому результат данной работы - просто оптимизированный алгоритм F4.

8 Оптимизация имплементации алгоритма F4

Даже после всех упомянутых выше математических оптимизаций, было видно что код неоптимален. Об оптимизациях кода поговорим именно в этой секции.

8.1 TermHash

На первых стадиях, все структуры для поиска одинаковых элементов были реализованы через **std::set**. **std::set** — это красно-черное бинарное дерево. Проблема с $Term$ - ordering. Как устроен поиск в бинарном дереве - спускаемся от корневой вершины либо влево, либо вправо, в зависимости от того, больше мы текущего значения, или нет. Для этого сравнения,

[4] https://github.com/KaurkerDevourer/ff4/pull/14

в любом ордеринге нам нужно пройти по всем элементам в $term$. Таким образом, ассимптотика на самом деле $O(logN \cdot term.size())$. Поэтому выбор был сделан в сторону хеш таблицы. Для этого была реализована базовая хеш функция, перфоманс улучшился на 15%.

8.2 TermRef

Однако после реализации хеша и перехода на хеш таблицы возникла новая проблема. Хеш таблицы реализованы так, что сначала они сравнивают хеш, а потом сами ключи. Поэтому они хранят копию ключа. **Term** - достаточно дорогой объект для копирования. Поэтому пришлось завести **TermRef**. Обычного указателя недостаточно, так как нужен кастомным operator==, чтобы сравнивать именно внутренности терма, а не то, на что указывает указатель. Это не сильно ускорило код (пару процентов), зато мы сделали максимально аккуратно, а главное nocopy.

8.3 CriticalPair

Внутри left и right лежат как ссылки. Достичь этого было не очень просто - есть много критериев, удалений, нужно внимательно следить за памятью. Такой подход дал большое ускорение, так как полиномы очень тяжелые.

9 Benchmarks

9.1 Профилирование

Во время написания библиотеки, много времени было уделено на профайлинг. Приведем тут некоторые примеры. Итоговый профиль выглядит так:

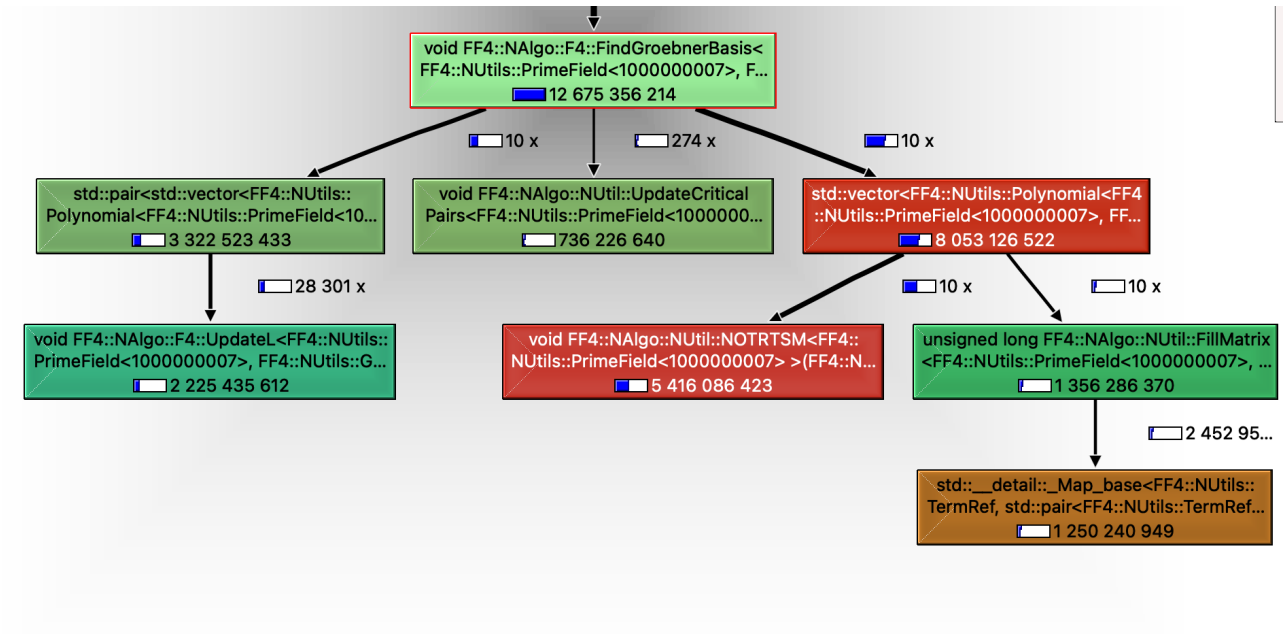

Рис. 9.1: Профиль ff4::F4

Для этого конкретного профиля использовался тест katsura-10. 25% занимает **SymbolicPreprocessing**. 65% занимают матричные вычисления. Из них - 9.5% занимают хеш мапы, 43% занимает **NOTRSM**. Если же использовать **GBLA**, то получается следующая картина:

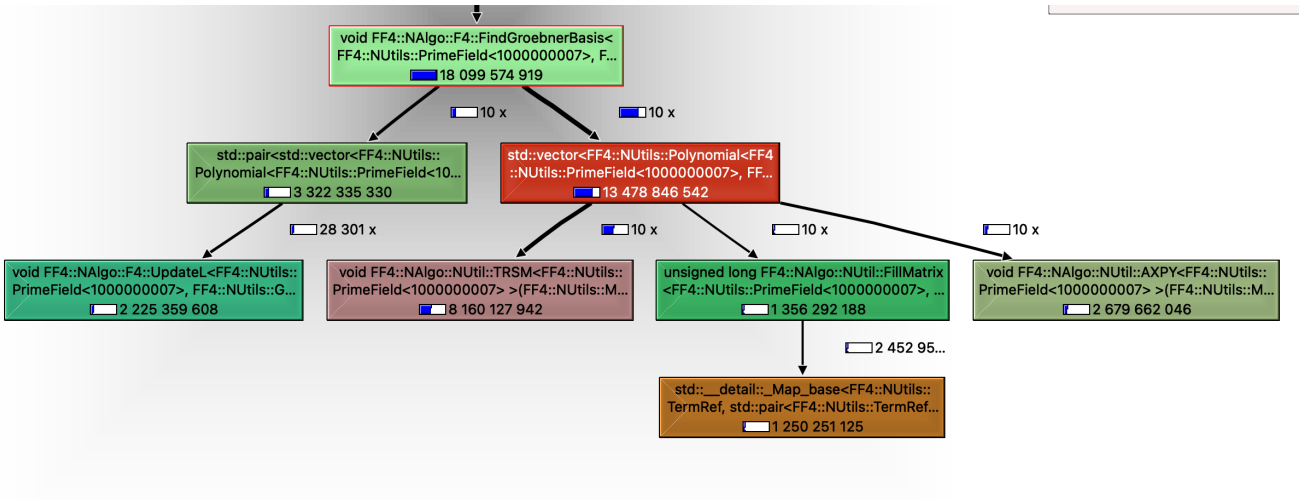

Рис. 9.2: Профиль ff4::F4 с GBLA

А вот так выглядил профиль до замены всех map на hash_map:

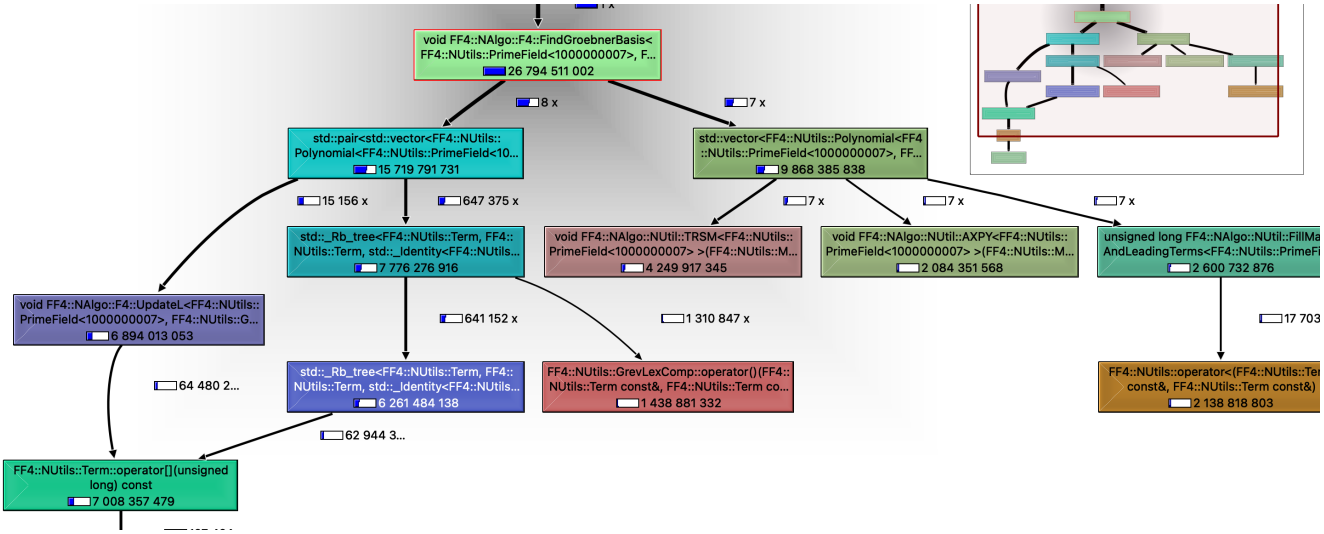

Рис. 9.3: Профиль ff4::F4 с map вместо hash_map

Исходя из профайлинга можно сделать вывод, что только оптимизаций матричных вычислений недостаточно. В любой реализации, **SymbolicPreprocessing** занимает достаточно большую часть времени. Это еще один аргумент в пользу того, что **Simplify** метод переоценен, и на самом деле не нужен.

9.2 Результаты

Тест	ff4::F4	openf4	GroebnerBasisLib::F4
sym3-3	123 мкс	474 мкс	2 мс
cyclic4	5.9 мкс	523 мкс	1.6 мс
cyclic5	363 мкс	1.3 мс	114 мс
cyclic6	3 мс	7.1 мс	1.7 с
cyclic7	73 мс	231 мс	—
katsura4	4 мкс	472 мкс	2.1 мс
katsura5	524 мкс	791 мкс	4.7 мс
katsura9	127 мс	106 мс	—
katsura10	641 мс	586 мс	—
katsura11	3.5 с	3.5 с	—
katsura12	21.3 с	22.4 с	—

Таблица 9.1: Измерения производительности алгоритмов F4 для $p = 10^9 + 7$, grevlex

По ним заметно, что мы обгоняем **openf4** на всех тестах кроме тестов среднего размера семейства **katsura**. На самом деле, на самых маленьких тестах мы тоже не быстрее производим вычисления, но интерфейс **openf4** вводит его в рамки — ему приходится парсить строки, и из них составлять полиномы, что на маленьких тестах занимает большую часть времени. Интересно, что на **katsura12** мы уже быстрее **openf4**. Это говорит о том, что наш подход очень удачный — чем больше входные данные, тем мы ближе к **openf4**, а потом и вовсе быстрее.

ff4::ImprovedBuchberger на **cyclic7** работает 23.6 секунды, а на **katsura9** и вовсе ООМится.

10 Итоги

По результатам работы, мы оптимизировали алгоритм F4 настолько, что он стал работать быстрее стандартных open-source вариантов. Кроме того, мне не удалось найти ни одного open-source проекта с реализацией алгоритма F4, который работал бы быстрее приведенного в работе. Также, стоит заметить, что ff4 позволяет изменять поле и порядок, чем, не может похвастаться большинство библиотек, например, openf4.

Ещё одним из результатов работы является NOGBLA, который показал, что иногда эффективнее использовать нетрадиционные методы, и тот факт, что метод Simplify сильно переоценён многими авторами.

Список литературы

[1] Bruno Buchberger. “Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal”. B: *Ph. D. Thesis, Math. Inst., University of Innsbruck* (1965).

[2] Thomas W Dubé. “The structure of polynomial ideals and Gröbner bases”. B: *SIAM Journal on Computing* 19.4 (1990), с. 750—773.

[3] Ernst W Mayr и Albert R Meyer. “The complexity of the word problems for commutative semigroups and polynomial ideals”. B: *Advances in mathematics* 46.3 (1982), с. 305—329.

[4] Rüdiger Gebauer и H Michael Möller. “On an installation of Buchberger’s algorithm”. B: *Journal of Symbolic computation* 6.2-3 (1988), с. 275—286.

[5] Jean-Charles Faugere. “A new efficient algorithm for computing Grobner basis”. B: *(F4)* (2002).

[6] Dylan Peifer. “The F4 Algorithm”. B: (2017).

[7] Brice Boyer, Christian Eder, Jean-Charles Faugere, Sylvian Lachartre и Fayssal Martani. “GBLA: Gröbner basis linear algebra package”. B: *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*. 2016, с. 135—142.

[8] Christian Eder. *Computing Gröbner Bases – a short overview*. Technische Universität Kaiserslautern, 11 сент. 2014. URL: https://caramba.loria.fr/sem-slides/201409111030.pdf.

[9] John Perry. “An extension of Buchberger’s criteria for Gröbner basis decision”. B: *LMS Journal of Computation and Mathematics* 13 (2010), с. 111—129.

[10] Hoon Hong и John Perry. “Are Buchberger’s criteria necessary for the chain condition?” B: *Journal of Symbolic Computation* 42.7 (2007), с. 717—732.

[11] Hoon Hong и John Perry. “Corrigendum to? Are Buchberger? s criteria necessary for the chain condition??[J. Symbolic Comput. 42 (2007) 717? 732]”. B: *Journal of Symbolic Computation* 43.3 (2008), с. 233.

[12] Till Stegers. “Faugere’s F5 algorithm revisited”. B: *Cryptology ePrint Archive* (2006).

[13] Jean-Pierre Merlet. *Polynomial systems*. URL: https://www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/node1.html.

A Семейства примеров

Я специально не определял, что это за семейства тестов мы используем. Нужно это для погружения в атмосферу, которой окружены базисы Грёбнера. Дело в том, что если вы проведете какое-то время изучая работы по базисам Грёбнера, вы наткнетесь на семейства тестов **katsura**, **cyclic** и другие. И очень сложно разобраться, откуда же берутся данные примеры и как их генерировать. Со второй проблемой помогают справиться специально написанные в этой работе генераторы[5]. Первая проблема гораздо серьезнее. Например, для **cyclic** даже интуитивно понятно, что это и как его строить:

$$C_n(\mathbf{x}) = \begin{cases} x_0 + x_1 + \cdots + x_{n-1} = 0 \\ x_0x_1 + x_1x_2 + \cdots + x_{n-2}x_{n-1} + x_{n-1}x_0 = 0 \\ i = 3, 4, \ldots, n-1 : \sum_{j=0}^{n-1} \prod_{k=j}^{j+i-1} x_{k \bmod n} = 0 \\ x_0x_1x_2 \cdots x_{n-1} - 1 = 0. \end{cases}$$

Рис. A.1: Cyclic

[5] https://github.com/KaurkerDevourer/ff4/tree/main/generators

Однако для **katsura**, непонятно уже ничего. Проблема семейств тестов - нет никакой общей базы. Лучшее что я смог найти, это примеры конкретных тестов в книге посвященной алгоритму F5: [12]. Важно понимать, что под h во всех тестах имеется ввиду константа. Тем не менее, так как книга старая, то многие ссылки уже не работают. Зная, как тяжело найти определение семейства **katsura**, очень хочется поделиться им в этой работе. Само определение находится на [13], и выглядит следующим образом:

$$\text{for } m \in \{-n+1, \ldots, n-1\}$$
$$\sum_{l=-n}^{l=n} u(l)u(m-l) = u(m)$$
$$\sum_{l=-n}^{l=n} u(l) = 1$$
$$u(l) = u(-l)$$
$$u(l) = 0 \text{ for } |l| > n$$

Рис. A.2: Katsura

Чтобы осознать что это за тесты, в любом случае придется немного посидеть с ручкой и бумажкой. Отрицательные числа тут определены, только для того, чтобы дать красивую формулу. На самом деле, тут отражается влево изначальный набор коэффициентов $[0, 1, 2, 3] \rightarrow [3, 2, 1, 0, 1, 2, 3]$. Каждый полином состоит из произведения термов, у которых разница в индексах ровно l. Если посмотреть в приведенный выше генератор станет еще понятнее.

Надеюсь, этот кусочек поможет лучше разобраться с семействами тестов в будущих работах.