Содержание

Введение

Теория помехоустойчивого кодирования является активно развивающейся областью науки и техники второй половины двадцатого, начала двадцать первого века. Её задача была определена Клодом Шенноном в 1948 году в статье «Математическая теория связи» [1], где были сформулированы основные составляющие модели передачи данных по каналу с шумом. Алгебраическая теория кодирования, сформировавшаяся в 60–70-е гг., решает задачу борьбы с помехами в канале алгебраическими методами. К 80-м годам двадцатого века были открыты многочисленные классы помехоустойчивых кодов, некоторые из которых нашли широкое применение в практике; из последних стоит отметить коды Рида–Соломона. В то же время теория кодирования пережила, наверное, самый большой переворот за свою историю связанный с применением алгебро-геометрических методов в конструировании кодов.

Первым заметившим, что аппарат алгебраической геометрии можно использовать для создания классов кодов с хорошими (не достигавшимися ранее) параметрами, был Гоппа [2]. Стоит также отметить фундаментальную работу в этом направлении российских учёных Влэдуца и Цфасмана [3] 1982 года. С конца 80-х и на протяжении 90-х количество работ, посвященных этим вопросам, только возрастало. Упомянем серию публикаций группы датских учёных по конструированию и декодированию одного класса сравнительно простых с точки зрения использовавшихся фактов алгебраической геометрии кодов [16] [17] [28]. Надо признать, что в 2000-е центр внимания профессионального сообщества сместился в сторону списочного декодирования.

С наступлением «алгебро-геометрической эры» в кодировании стал вопрос о том, какие из классических подходов могут быть применены в новых условиях. Оказалось, что к таковым относится конструкция кодов Рида—Соломона и общий метод их декодирования, включая такой конструктивный элемент последнего как алгоритм Берлекэмпа—Месси (BM-алгоритм). BM-алгоритм с момента своего появления в 1968 году получил ши-

рокое распространение не только для декодирования разнообразных классов кодов теории помехоусточивого кодирования, но и в других областях прикладной (например, криптография [4]) и фундаментальной (например, теория аппроксимаций Паде [11]) математики.

Обобщение BM-алгоритма для декодирования алгебро-геометрических кодов связано с увеличением размерности задачи. Такое обобщение предложил Саката [14], после чего алгоритм стал именоваться алгоритмом Берлекэмпа—Месси—Сакаты (BMS-алгоритмом). Он был применён для декодирования семейства кодов из [16], кроме того были разработаны более общие коды, для декодирования которых также может быть использован BMS-алгоритм [5].

Одной из проблем в области помехоустойчивого кодирования, основанного на фактах алгебраической геометрии, является сложность реализации рассматриваемых конструкций сравнительно с классическими (реализации как программной, так и, в особенности, аппаратной).

Целью данной работы стала реализация BMS-алгоритма и изложение подхода к его применению в декодировании АГ-кодов. Для достижения этой цели были поставлены задачи:

- изучение BMS-алгоритма, его связи с BM-алгоритмом и возможностей его реализации;
- разработка схемы BMS-алгоритма, которая может служить практическим руководством к реализации;
- реализация BMS-алгоритма с использованием разработанной схемы;
- изучение АГ-кодов типа кодов Рида—Соломона и метода их декодирования [16] [17] [28];
- создание схемы декодера АГ-кодов типа кодов Рида—Соломона.

Результаты работы докладывались на конференции «Неделя науки» ЮФУ и X Международной научно-практической конференции «Информационная безопасность» в г. Таганроге, тезисы опубликованы в соответству-

ющих сборниках [6] [7]. Часть результатов опубликована в издании, входящем в список ВАК [8].

Работа состоит из введения, трёх глав и приложения, содержащего исходные коды выполненной реализации BMS-алгоритма.

1 BMS-алгоритм

1.1 BM-алгоритм

Первоначальная версия алгоритма Берлекэмпа—Месси (BM-алгоритма) была изложена Берлекэмпом в 1968 году [9] в качестве элемента конструкции декодера кодов Боуза—Чоудхудри—Хоквингема над конечным полем. Хотя в этой работе была указана возможность формулировки решаемой задачи с использованием понятия линейного регистра сдвига с обратной связью, алгоритм описывался исключительно в терминах полиномов и был весьма сложен для понимания. Спустя год Месси [10] предложил свою интерпретацию алгоритма, который теперь строил линейный регистр сдвига минимальной длины, генерирующий заданную последовательность элементов конечного поля. Эта интерпретация оказалась полезной для более широкого распространения алгоритма, получившего название по имени этих двух ученых.

С момента появления алгоритма вышло немало работ, развивающих, обобщающих и по-новому интерпретирующих его идеи (например, [11] [13] [25]). Рассматриваемый алгоритм находит применение при декодировании различных классов кодов: кодов Рида—Соломона, кодов БЧХ, циклических и обобщенных циклических кодов, альтернантных кодов и кодов Гоппы, и, наконец, наиболее общего и актуального на сегодня класса кодов — алгебро-геометрических кодов (вернее, некоторых их подклассов).

Построим структурную схему BM-алгоритма, следуя его описанию в [12].

Последовательностью над полем $\mathbb{F}_q$ будем называть любую функцию $u : \mathbb{N}_0 \to \mathbb{F}_q$, заданную на множестве целых неотрицательных чисел и принимающую значения в этом поле.

Элементы последовательности u будут обозначаться $u(i)$. Будет встречаться также понятие *отрезка последовательности*, которое получается естественным образом из ограничения функции, упомянутой в определении.

Последовательность u будем называть *линейной рекуррентной после-*

довательностью (ЛРП) *порядка* $m > 0$ *над полем* $\mathbb{F}_q$, если существуют константы $f_0, \ldots, f_{m-1} \in \mathbb{F}_q$ такие, что

$$u(i+m) = \sum_{j=0}^{m-1} f_j \cdot u(i+j), i \geq 0.$$

Указанное выражение назовем *законом рекурсии* или *линейным рекуррентным соотношением*. Говорят, что $\{f_j\}_{j=0}^{m-1}$ задают закон рекурсии для ЛРП u.

Как видно, первые m элементов последовательности не связаны какими-либо ограничениями — они имеют особое значение, их, как правило, называют *начальным отрезком* последовательности u.

Пусть u — ЛРП, для которой $\{f_j\}_{j=0}^{m-1}$ задают закон рекурсии. Многочлен:

$$F(x) = x^m - \sum_{j=0}^{m-1} f_j \cdot x^j$$

с коэффициентами из поля $\mathbb{F}_q$ назовем *характеристическим многочленом* ЛРП u.

Таким образом, каждой ЛРП можно поставить в соответствие характеристический многочлен и обратно, каждому нормированному многочлену можно поставить в соответствие ЛРП. Можно показать однако, что одна и та же последовательность может задаваться *разными* законами рекурсии и, соответственно, иметь разные характеристические полиномы.

Характеристический полином ЛРП u, имеющий наименьшую степень, назовем её *минимальным многочленом*, а его степень — *линейной сложностью* ЛРП u.

Минимальные многочлены ЛРП, а также их линейная сложность, являются важными характеристиками ЛРП.

Пусть u — последовательность над полем $\mathbb{F}_q$. Обозначим через

$$u\left(\overline{0, l-1}\right) = (u(0), \ldots, u(l-1))$$

начальный отрезок u. Будем говорить, что многочлен

$$G(x) = x^m - \sum_{j=0}^{m-1} b_j \cdot x^j$$

вырабатывает отрезок $u\left(\overline{0, l-1}\right)$, если

$$\forall i \in [0, l-m-1] : \; u(i+m) = \sum_{j=0}^{m-1} b_j \cdot u(i+j),$$

то есть если данный отрезок последовательности является отрезком некоторой ЛРП с характеристическим многочленом $G(x)$.

Естественным образом определяется понятие линейной сложности отрезка последовательности как минимальной степени из всех полиномов, вырабатывающих данный отрезок.

Алгоритм Берлекэмпа—Месси строит многочлен $G(x)$ наименьшей степени, вырабатывающий отрезок $u\left(\overline{0, l-1}\right)$. Чтобы перейти к непосредственному описанию алгоритма, требуется ввести ещё ряд вспомогательных определений.

Введём операцию умножения произвольного многочлена

$$H(x) = \sum_{j=0}^{n} h_j x^j$$

на любую последовательность v, результатом которой будет последовательность w, такая что:

$$(H(x) \cdot v)(i) = w(i) \stackrel{\text{def}}{=} \sum_{j=0}^{n} h_j \cdot v(i+j)$$

Очевидно, операция является линейной относительно полинома, входящего в неё.

Для нормированного полинома $G(x)$ определим параметры:

1. $k_u(G)$ — количество лидирующих нулей последовательности $G(x) \cdot u$

или ∞, если эта последовательность нулевая.

2. $l_u(G) = k_u(G) + \deg(G)$.

Легко убедиться, что $l_u(G)$ — максимальная длина начального отрезка u, вырабатываемого $G(x)$. Действительно, пусть

$$G(x) = \sum_{j=0}^{m} g_j \cdot x^j = x^m - \sum_{j=0}^{m-1} b_j \cdot x^j.$$

Обозначим $G(x) \cdot u = v$. Тогда:

$$\forall i \in [0, l_u(G) - m - 1] : \ v(i) = 0,$$

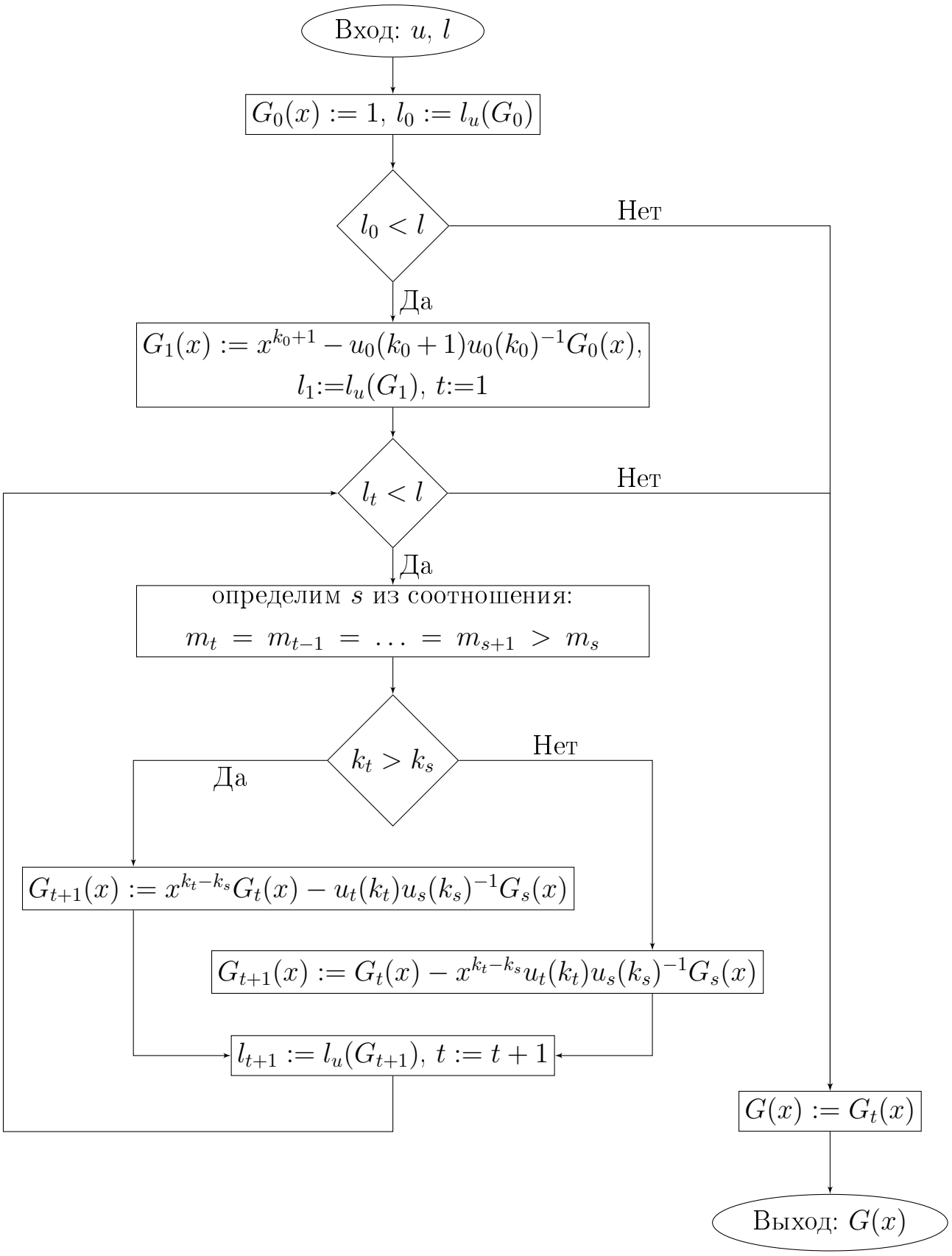
но:

$$0 = v(i) = \sum_{j=0}^{n} g_j \cdot u(i+j) = u(i+m) - \sum_{j=0}^{m-1} b_j \cdot u(i+j),$$

что и дает искомое:

$$\forall i \in [0, l_u(G) - m - 1] : u(i+m) = \sum_{j=0}^{m-1} b_j \cdot u(i+j).$$

Теперь можно привести полную схему классического ВМ-алгоритма. Зададимся последовательностью u над полем $\mathbb{F}_q$ и числом l. Найдем минимальный полином $G(x)$, вырабатывающий отрезок $u\left(\overline{0, l-1}\right)$, используя ВМ-алгортим, описываемый нижеследующей структурной схемой, построенной по работе [12].


Вход: u, l
$G_0(x) := 1$, $l_0 := l_u(G_0)$
$l_0 < l$
Нет
Да
$G_1(x) := x^{k_0+1} - u_0(k_0+1)u_0(k_0)^{-1}G_0(x)$,
$l_1{:=}l_u(G_1)$, $t{:=}1$
$l_t < l$
Нет
Да
определим s из соотношения:
$m_t = m_{t-1} = \ldots = m_{s+1} > m_s$
$k_t > k_s$
Нет
Да
$G_{t+1}(x) := x^{k_t-k_s}G_t(x) - u_t(k_t)u_s(k_s)^{-1}G_s(x)$
$G_{t+1}(x) := G_t(x) - x^{k_t-k_s}u_t(k_t)u_s(k_s)^{-1}G_s(x)$
$l_{t+1} := l_u(G_{t+1})$, $t := t+1$
$G(x) := G_t(x)$
Выход: $G(x)$

1.2 Предварительные обозначения и конструкции для BMS-алгоритма

Введём обозначения для следующих множеств: $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$, $\Sigma_{\mathbf{0}} = \mathbb{N}_0^2$. Элементы $\Sigma_{\mathbf{0}}$ будем называть точками и выделять полужирным шрифтом, например: $\boldsymbol{n} \in \Sigma_{\mathbf{0}}$. Компоненты точек будем обозначать нижними индексами, например, компоненты точки $\boldsymbol{n}$: n_1, n_2. Для точек $\Sigma_{\mathbf{0}}$ определено покомпонентное сложение:

$$\forall \boldsymbol{n}, \boldsymbol{m} \in \Sigma_{\mathbf{0}} : \boldsymbol{n} + \boldsymbol{m} \stackrel{\text{def}}{=} (n_1 + m_1, n_2 + m_2).$$

Аналогично будем использовать вычитание точек, когда оно корректно (обе координаты уменьшаемого не меньше соответствующих координат вычитаемого).

Сумма точки и множества точек определяется так:

$$\forall s \in \Sigma_{\mathbf{0}}\, \forall M \subset \Sigma_{\mathbf{0}} :\ s + M \stackrel{\text{def}}{=} \{s + m \mid m \in M\}.$$

Введём два отношения порядка на $\Sigma_{\mathbf{0}}$:

1. $\forall \boldsymbol{m}, \boldsymbol{n} \in \Sigma_{\mathbf{0}} :\ \boldsymbol{m} < \boldsymbol{n}$ тогда и только тогда, когда

$$(m_1 \leqslant n_1) \wedge (m_2 \leqslant n_2) \wedge (\boldsymbol{m} \neq \boldsymbol{n});$$

2. $\forall \boldsymbol{m}, \boldsymbol{n} \in \Sigma_{\mathbf{0}} :\ \boldsymbol{m} <_{\mathrm{T}} \boldsymbol{n}$ тогда и только тогда, когда

$$\big(m_1 + m_2 < n_1 + n_2\big) \ \vee\ \big((m_1 + m_2 = n_1 + n_2) \wedge (m_2 < n_2)\big). \qquad (1)$$

Первое отношение является отношением частичного порядка, а второе — отношением линейного порядка. Естественным образом определяются рефлексивные версии этих отношений («нестрогие неравенства»): $\leqslant, \leqslant_{\mathrm{T}}$. Будут использоваться также обозначения $\boldsymbol{m} \not< \boldsymbol{n}$ и $\boldsymbol{m} \not\leqslant \boldsymbol{n}$, когда выполнено одно из неравенств $m_1 > n_1$ или $m_2 > n_2$, или оба сразу.

Линейный порядок позволяет для каждой точки $\boldsymbol{n}$ единственным образом определить *непосредственно следующую за ней* точку $\boldsymbol{n'}$:

$$\forall \boldsymbol{n} \in \Sigma_{\mathbf{0}} : \ \boldsymbol{n'} = \begin{cases} (n_1 - 1, n_2 + 1), & \text{если } n_1 > 0, \\ (n_2 + 1, 0), & \text{если } n_1 = 0. \end{cases}$$

Введём множества:

$$\begin{aligned} \forall \boldsymbol{m} \in \Sigma_{\mathbf{0}} : \quad & \Sigma_{\boldsymbol{m}} \stackrel{\text{def}}{=} \{\boldsymbol{p} \in \Sigma_{\mathbf{0}} \mid \boldsymbol{m} \leq \boldsymbol{p}\}; \\ \forall \boldsymbol{m}, \boldsymbol{n} \in \Sigma_{\mathbf{0}} : \quad & \Sigma_{\boldsymbol{m}}^{\boldsymbol{n}} \stackrel{\text{def}}{=} \{\boldsymbol{p} \in \Sigma_{\mathbf{0}} \mid (\boldsymbol{m} \leq \boldsymbol{p}) \wedge (\boldsymbol{p} <_{\mathrm{T}} \boldsymbol{n})\}. \end{aligned}$$

Отметим, что если $\boldsymbol{n} \leqslant_{\mathrm{T}} \boldsymbol{m}$, то множество $\Sigma_{\boldsymbol{m}}^{\boldsymbol{n}}$ пусто. Кроме того, обратим особое внимание, что $\boldsymbol{n} \notin \Sigma_{\boldsymbol{m}}^{\boldsymbol{n}}$.

Пусть задано поле Галуа $\mathbb{F}_q$. Полином от двух переменных над полем $\mathbb{F}_q$, т. е. элемент кольца $\mathbb{F}_q[x] \stackrel{\text{def}}{=} \mathbb{F}_q[x_1, x_2]$, будем записывать так:

$$f(x) = \sum_{\boldsymbol{m} \in \Gamma_f} f_{\boldsymbol{m}} \cdot x^{\boldsymbol{m}},$$

где $x^{\boldsymbol{m}} = x_1^{m_1} \cdot x_2^{m_2}$, $f_{\boldsymbol{m}} \in \mathbb{F}_q$, а конечное множество

$$\Gamma_f = \{\boldsymbol{m} \in \Sigma_{\mathbf{0}} \mid f_{\boldsymbol{m}} \neq 0\}$$

— носитель множества коэффициентов. Линейный порядок на $\Sigma_{\mathbf{0}}$ позволяет корректно определить (*старшую*) *степень* полинома f, которую будем обозначать $\mathrm{LP}(f)$:

$$\forall f \in \mathbb{F}_q[x] : \ \mathrm{LP}(f) \stackrel{\text{def}}{=} \max_{\boldsymbol{m} \in \Gamma_f} \boldsymbol{m},$$

где максимум берется в смысле линейного порядка на $\Sigma_{\mathbf{0}}$. Для произвольного (упорядоченного) набора полиномов $\mathcal{F} = \{f^{(i)}(x)\}_{i=1}^{l}$ определим:

$$\mathrm{LP}(\mathcal{F}) \stackrel{\text{def}}{=} \{\boldsymbol{s}^{(i)} = \mathrm{LP}(f^{(i)})\}_{i=1}^{l}$$

Конечной двумерной последовательностью (или просто *последователь-*

ностью) u «длины» $\boldsymbol{p} \in \Sigma_{\boldsymbol{0}}$ над полем $\mathbb{F}_q$ назовем отображение

$$u : \Sigma_{\boldsymbol{0}}^{\boldsymbol{p}} \to \mathbb{F}_q.$$

Ограничение этого отображения на множество $\Sigma_{\boldsymbol{0}}^{\boldsymbol{n}}$ для некоторого $\boldsymbol{n} \in \Sigma_{\boldsymbol{0}}^{\boldsymbol{p}'}$ обозначим $u^{\boldsymbol{n}}$ и назовем $\boldsymbol{n}$-*срезкой* u. Отметим, что $\boldsymbol{n}$-срезка u в свою очередь является двумерной последовательностью длины $\boldsymbol{n}$.

Для полинома f степени $\boldsymbol{s}$, последовательности u длины $\boldsymbol{p}$ и точки $\boldsymbol{n} \in \Sigma_{\boldsymbol{s}}^{\boldsymbol{p}}$ определим элемент поля $\mathbb{F}_q$:

$$f[u]_{\boldsymbol{n}} \stackrel{\text{def}}{=} \sum_{\boldsymbol{m} \in \Gamma_f} f_{\boldsymbol{m}} \cdot u_{\boldsymbol{m}+\boldsymbol{n}-\boldsymbol{s}}. \tag{2}$$

Напомним, что в случае $\boldsymbol{p} \leqslant_{\mathrm{T}} \boldsymbol{s}$ множество $\Sigma_{\boldsymbol{s}}^{\boldsymbol{p}}$ пусто. Будем писать $f \in \mathrm{VALPOL}(u)$ тогда и только тогда, когда

$$\forall \boldsymbol{n} \in \Sigma_{\boldsymbol{s}}^{\boldsymbol{p}} : \; f[u]_{\boldsymbol{n}} = 0. \tag{3}$$

Таким образом, в случае $\boldsymbol{p} \leqslant_{\mathrm{T}} \boldsymbol{s}$ условие $f \in \mathrm{VALPOL}(u)$ выполнено тривиально.

Очевидно, для любой последовательности u длины p выполнено:

$$\forall \boldsymbol{m}, \boldsymbol{n} \in \Sigma_{\boldsymbol{0}}^{\boldsymbol{p}'} : \; m <_{\mathrm{T}} n \Rightarrow \mathrm{VALPOL}(u^{\boldsymbol{m}}) \supset \mathrm{VALPOL}(u^{\boldsymbol{n}})$$

Уточним этот факт для случая последовательно идущих точек:

$$\forall \boldsymbol{n} \in \Sigma_{\boldsymbol{0}}^{\boldsymbol{p}} \, \forall f \in \mathrm{VALPOL}(u^{\boldsymbol{n}}) : \\ f \in \mathrm{VALPOL}(u^{\boldsymbol{n}'}) \Leftrightarrow \big(\boldsymbol{n} \notin \Sigma_{\mathrm{LP}(f)}^{\boldsymbol{n}'}\big) \vee \big(f[u]_{\boldsymbol{n}} = 0\big).$$

Действительно, $\forall \boldsymbol{s}, \boldsymbol{n} \in \Sigma_{\boldsymbol{0}} : \; \Sigma_{\boldsymbol{s}}^{\boldsymbol{n}'} \setminus \Sigma_{\boldsymbol{s}}^{\boldsymbol{n}} \subset \{\boldsymbol{n}\}$. Таким образом, надо рассмотреть два случая:

1. $\Sigma_{\mathrm{LP}(f)}^{\boldsymbol{n}'} \setminus \Sigma_{\mathrm{LP}(f)}^{\boldsymbol{n}} = \emptyset$, т. е. $\Sigma_{\mathrm{LP}(f)}^{\boldsymbol{n}'} = \Sigma_{\mathrm{LP}(f)}^{\boldsymbol{n}}$, тогда по определению

$$f \in \mathrm{VALPOL}(u^{\boldsymbol{n}}) \Rightarrow f \in \mathrm{VALPOL}(u^{\boldsymbol{n}'}).$$

2. $\Sigma^{\boldsymbol{n}'}_{\mathrm{LP}(f)} \setminus \Sigma^{\boldsymbol{n}}_{\mathrm{LP}(f)} = \{\boldsymbol{n}\}$, тогда

$$f \in \mathrm{VALPOL}(u^{\boldsymbol{n}'}) \Leftrightarrow$$
$$\forall m \in \Sigma^{\boldsymbol{n}'}_{LP(f)} = \Sigma^{\boldsymbol{n}}_{LP(f)} \cup \{\boldsymbol{n}\} : \ f[u]_{\boldsymbol{m}} = 0 \Leftrightarrow$$
$$f \in \mathrm{VALPOL}(u^{\boldsymbol{n}}) \wedge f[u]_{\boldsymbol{n}} = 0. \quad (4)$$

Назовём набор точек $\{\boldsymbol{s}^{(i)}\}_{i=1}^{l}$ *набором гиперболического типа* (или просто *гиперболическим набором*), если для него выполнены соотношения:

$$s_1^{(1)} > s_1^{(2)} > \ldots > s_1^{(l)} = 0, \ 0 = s_2^{(1)} < s_2^{(2)} < \ldots < s_2^{(l)}. \quad (5)$$

Каждому гиперболическому набору точек $\{\boldsymbol{s}^{(i)}\}_{i=1}^{l}$ поставим в соответствие множество $\Delta = \Delta(\{\boldsymbol{s}^{(i)}\}_{i=1}^{l})$, определяемое по формуле:

$$\Delta(\{\boldsymbol{s}^{(i)}\}_{i=1}^{l}) = \Sigma_{\boldsymbol{0}} \setminus (\Sigma_{\boldsymbol{s}^{(1)}} \cup \Sigma_{\boldsymbol{s}^{(2)}} \cup \ldots \cup \Sigma_{\boldsymbol{s}^{(1)}}),$$

Множество $\Delta = \Delta(\{\boldsymbol{s}^{(i)}\}_{i=1}^{l})$ в этом случае назовем Δ-*множеством* для набора $\{\boldsymbol{s}^{(i)}\}_{i=1}^{l}$. В свою очередь набор $\{\boldsymbol{s}^{(i)}\}_{i=1}^{l}$ называется *определяющими точками* Δ.

Если для некоторого упорядоченного набора полиномов

$$\mathcal{F} = \{f^{(i)}(x)\}_{i=1}^{l}$$

их степени

$$\mathrm{LP}(\mathcal{F}) = \{\boldsymbol{s}^{(i)}\}_{i=1}^{l}$$

составляют гиперболический набор, то $\Delta(\mathrm{LP}(\mathcal{F}))$ мы будем обозначать просто $\Delta(\mathcal{F})$.

Пусть дана конечная двумерная последовательность u «длины» $\boldsymbol{p}$. *Минимальным множеством* (полиномов) для последовательности u называется набор $\mathcal{F} = \{f^{(i)}(x)\}_{i=1}^{l}$, удовлетворяющий условиям:

1. $\mathcal{F} \subset \mathrm{VALPOL}(u)$.

2. $\mathrm{LP}(\mathcal{F})$ — гиперболический набор.

3. $\forall g \in \mathbb{F}_q[x] :\ g \in \mathrm{VALPOL}(u) \Rightarrow \mathrm{LP}(g) \notin \Delta(\mathcal{F})$.

Условие (3) гарантирует единственность гиперболического набора, задаваемого любым минимальным множеством для данной последовательности. Если $\mathcal{F}$ — некоторое минимальное множество для последовательности u, то $\Delta(\mathcal{F})$ можно обозначить $\Delta(u)$. Существование минимального множества для последовательности будет доказано конструктивно — описанием алгоритма, строящего его.

1.3 Схема BMS-алгоритма

Данный алгоритм, разработанный Сакатой в [14] на основе модификации классического алгоритма Берлекэмпа–Месси, строит минимальное множество полиномов F для произвольной последовательности u «длины» $\boldsymbol{p}$. Перед тем как описать шаги алгоритма, нужно ввести некоторые множества, с которыми работает алгоритм, и дополнительные обозначения.

Алгоритм имеет итеративный характер: на каждой итерации значение параметра $\boldsymbol{n} \in \Sigma_{\mathbf{0}}$ («мультиномер итерации») заменяется на точку, непосредственно следующую за $\boldsymbol{n}$. Как только $\boldsymbol{n}$ станет равным $\boldsymbol{p}$ (длине последовательности), алгоритм завершится.

К началу $\boldsymbol{n}$-ой итерации сформированы:

$F = \{f^{(i)}(x)\}_{i=1}^{l}$ — минимальное множество для $\boldsymbol{n}$-срезки u;

$G = \{g^{(i)}(x)\}_{i=1}^{l-1}$ — вспомогательное множество полиномов, такое что:

$$\forall i \in [1, l-1]\ \exists \boldsymbol{p}^{(i)} \in \Sigma_{\mathbf{0}}^{\boldsymbol{n}} : g^{(i)} \in \mathrm{VALPOL}(u^{\boldsymbol{p}^{(i)}}) \setminus \mathrm{VALPOL}(u^{\boldsymbol{p}^{(i)'}}),$$

в частности, $d^{(i)} \stackrel{\text{def}}{=} g^{(i)}[u]_{\boldsymbol{p}^{(i)}} \neq 0$. Более точно: $g^{(i)}$ входил в F на итерации $\boldsymbol{p}^{(i)}$, но не мог входить в него при переходе к $\boldsymbol{p}^{(i)'}$ из-за того, что $d^{(i)} \neq 0$.

С F связано множество:

$$S = \{\boldsymbol{s}^{(i)}\}_{i=1}^{l} = \mathrm{LP}(F),$$

с G связаны три множества:

$$T = \{\boldsymbol{t}^{(i)}\}_{i=1}^{l-1} = \mathrm{LP}(G),$$
$$PG = \{\boldsymbol{p}^{(i)}\}_{i=1}^{l-1},$$
$$DG = \{d^{(i)}\}_{i=1}^{l-1}.$$

Введём следующие обозначения, которые используют перечисленные параметры алгоритма и, таким образом, зависят от мультиномера текущей итерации $\boldsymbol{n}$:

(1) Опишем число $\mathrm{inSD}(\boldsymbol{s}) \in [0, l-1]_{\mathbb{N}_0}$ для любого $\boldsymbol{s} \in \Sigma_{\mathbf{0}}$:

если $\boldsymbol{n} \notin \boldsymbol{s} + \Delta(F)$, положим $\mathrm{inSD}(\boldsymbol{s}) \stackrel{\text{def}}{=} 0$,

если $\boldsymbol{n} \in \boldsymbol{s} + \Delta(F)$, то $\exists i \in [1, l-1]_{\mathbb{N}_0}$, такой что:

$$(n_1 < s_1 + s_1^{(i)}) \wedge (n_2 < s_2 + s_2^{(i+1)}),$$

любой из таких i обозначим $\mathrm{inSD}(\boldsymbol{s})$.

Замечание 1 *inSD — «in shifted Delta-set», «в сдвинутом дельта-множестве»: мультиномер $\boldsymbol{n}$ текущей итерации находится в сдвинутом на $\boldsymbol{s}$ Δ-множестве.*

(2) $\forall i \in [1, l]_{\mathbb{N}}, j \in [0, l-1]_{\mathbb{N}_0}$:

$$\mathrm{BP}\langle i, j\rangle \stackrel{\text{def}}{=} \begin{cases} f^{(i)},\ j = 0; \\ x^{\boldsymbol{r}-\boldsymbol{s}^{(i)}} \cdot f^{(i)} - (d/d^{(j)}) x^{\boldsymbol{r}-\boldsymbol{n}+\boldsymbol{p}^{(j)}-\boldsymbol{t}^{(j)}} \cdot g^{(j)}, \text{ иначе,} \end{cases}$$

где $d = f^{(i)}[u]_{\boldsymbol{n}}$, $r_k = \max\{s_k^{(i)}; t_k^{(j)} + n_k - p_k^{(j)}\}$, $k \in \{1, 2\}$;

(3) $\forall i \in [1, l]_{\mathbb{N}}\ \forall k \in \{1, 2\}$:

$$\mathrm{SP}_k\langle i\rangle \stackrel{\text{def}}{=} x_k^{n_k - s_k^{(i)} + 1} \cdot f^{(i)};$$

Вход: двумерная последовательность u длины $\boldsymbol{p}$ над полем $\mathbb{F}_q$.

Выход: минимальное множество F для u.

Шаг 0. Положить $\boldsymbol{n} := (0,0)$, $F := \{1\}$ $(\Delta(F) = \emptyset)$, $G := DG := PG := \emptyset$.

Замечание 2 *Инициализация параметров алгоритма совпадает с [14, п. 5, шаг 1].*

Шаг 1. $\forall i \in [1,l]_{\mathbb{N}}$:

$$\textbf{if } \boldsymbol{s}^{(i)} \not\leqslant \boldsymbol{n} \ \vee f^{(i)}[u]_{\boldsymbol{n}} = 0 \textbf{ then } f^{(i)} \in F_V \textbf{ else } f^{(i)} \in F_N.$$

Замечание 3 *Множества $F_V, F_N \subset F$ вводятся в [14] после теоремы 1, п. 4. F_V — множество полиномов, которые не требуют изменения на текущей итерации (V — valid, эффективный, действительный). F_N — множество полиномов, которые необходимо модифицировать на данной итерации (N — nonvalid).*

Замечание 4 *Условие $\boldsymbol{s}^{(i)} \not\leqslant \boldsymbol{n}$ влечёт за собой $\boldsymbol{n} \notin \Sigma^{\boldsymbol{n}'}_{\boldsymbol{s}^{(i)}}$. Как указано в (4), этот факт вместе с имеющимся к началу $\boldsymbol{n}$-ой итерации $f^{(i)} \in \mathrm{VALPOL}(u^{\boldsymbol{n}})$ влечет за собой $f^{(i)} \in \mathrm{VALPOL}(u^{\boldsymbol{n}'})$. Этот случай не обсуждается в [14].*

Шаг 2. Введём $aux := \overline{0}_l$ — вектор из l компонент, пока нулевой.

$$\forall f^{(i)} \in F_N : aux[i] := \mathrm{inSD}(\boldsymbol{s}^{(i)}),$$

где $aux[i]$ — i-ая компонента вектора aux.

Шаг 3. **if** $\forall f^{(i)} \in F_N : \ aux[i] \neq 0$ **then**

$$\forall f^{(i)} \in F_N : \ f^{(i)} := \mathrm{BP}\langle i, aux[i]\rangle; \ \textbf{goto}\, \text{Шаг } 8.$$

Замечание 5 *Условие шага 3 взято непосредственно из описания алгоритма [14, п. 5, шаг 2]. Отличие состоит в использовании вспомогательного вектора aux, хранящего сразу все результаты проверок* $\boldsymbol{n} \in \mathrm{LP}(f^{(i)}) + \Delta$ *(выполняемых в этом месте в [14]) и использовании нашего обозначения* $\mathrm{inSD}(n)s$ *(шаг 2) для этих проверок.*

Замечание 6 *Формула для пересчёта* $f^{(i)}$ *вытекает из теоремы 1 в [14, п. 4], а верней, её обоснования, приведенного перед формулировкой. Использование теоремы 1 продиктовано [14, п. 5, шаг 2].*

Шаг 4. Построить следующие множества точек:

$$\Delta_1 = \{(s_1^{(i)}, s_2^{(i)}) \mid i \in [1, l]_{\mathbb{N}} \wedge \Big(f^{(i)} \in F_V \vee aux[i] \neq 0\Big)\};$$

$$\Delta_2 = \{(n_1 - s_1^{(i)} + 1, n_2 - s_2^{(i+1)} + 1) \mid i \in [1, l-1]_{\mathbb{N}} \wedge f^{(i)}, f^{(i+1)} \in F_N\};$$

$$\Delta_3 = \{(n_1 - s_1^{(i)} + 1, s_2^{(j)}) \mid i, j \in [1, l]_{\mathbb{N}} \wedge (f^{(i)}, f^{(j)} \in F_N) \wedge (\boldsymbol{s}^{(i)} + \boldsymbol{s}^{(j)} \leqslant \boldsymbol{n}) \wedge (\forall k \in (i, l]_{\mathbb{N}} : n_2 < s_2^{(k)} + s_2^{(j)})\};$$

$$\Delta_4 = \{(s_1^{(i)}, n_2 - s_2^{(j)} + 1) \mid i, j \in [1, l]_{\mathbb{N}} \wedge (f^{(i)}, f^{(j)} \in F_N) \wedge (\boldsymbol{s}^{(i)} + \boldsymbol{s}^{(j)} \leqslant \boldsymbol{n}) \wedge \forall k \in [1, j)_{\mathbb{N}} : n_1 < s_1^{(k)} + s_1^{(i)}\}.$$

$\Delta_{\text{new}} = \Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \Delta_4$.

Замечание 7 *Необходимость построения нового* Δ*-множества (а в след за ним и нового* F *— см. следующий шаг) указывается в [14,*

п. 5, шаг 2] со ссылкой на теорему 2. В теореме 2 описывается построение нового F по $\Delta_e(u^{\boldsymbol{n}'})$ — множеству «исключённых точек». Последнее составляют точки, которые обладают таким свойством: $\neg\exists f \in \mathrm{VALPOL}(u^{\boldsymbol{n}'}) : \mathrm{LP}(f) \in \Delta_e(u^{\boldsymbol{n}'})$. Фактически на нашем шаге 4 строится $\Delta_e(u^{\boldsymbol{n}'})$, но так как теорема 2 указывает способ построения полиномов, старшие степени которых являются определяющими точками $\Delta_e(u^{\boldsymbol{n}'})$, то $\Delta_e(u^{\boldsymbol{n}'}) = \Delta(u^{\boldsymbol{n}'})$.

Замечание 8 *Множество $\Delta_e(u^{\boldsymbol{n}})$ определяется в [14, стр. 327], основываясь на лемме 4. По построению видно, что $\Delta_e(u^{\boldsymbol{n}})$ это Δ-множество.*

Замечание 9 *Условия включения точек каждого из четырех типов описаны в [14]: Леммы 8 и 9 и предшествующие им рассуждения (после теоремы 1). Сами четыре типа перечислены в Лемме 7 [14].*

Замечание 10 *В соответствии с определением Δ-множества (см. (5)), при построении Δ_3 и Δ_4 можно рассматривать лишь $k = i + 1$ и $k = j - 1$ соответственно. Эта оптимизация не указана в [14].*

Замечание 11 *Информация об элементах Δ_r включает не только точку $\Sigma_{\mathbf{0}}$, но и «историю появления» этой точки, то есть числа i, j.*

Шаг 5. Построить следующие множества полиномов.

$F_1 = \{\mathrm{BP}\langle i, aux[i]\rangle \mid (s_1^{(i)}, s_2^{(i)}) \in \Delta_1\}$;

$$F_2 = \{\mathrm{BP}\langle k, i\rangle \mid (n_1 - s_1^{(i)} + 1, n_2 - s_2^{(i+1)} + 1) \in \Delta_2, \\ k : f^{(k)} \in F_N \ \wedge \ \boldsymbol{s}^{(k)} < \boldsymbol{t}\};$$

$$F_3 = \{\mathrm{BP}\langle j, i\rangle \mid (n_1 - s_1^{(i)} + 1, s_2^{(j)}) \in \Delta_3 \wedge i \neq l\} \cup \\ \{\mathrm{SP}_1\langle j\rangle \mid (n_1 + 1, s_2^{(j)}) \in \Delta_3\}$$

$$F_4 = \{\mathrm{BP}\langle i, j-1\rangle \mid (s_1^{(i)}, n_2 - s_2^{(j)} + 1) \in \Delta_4 \wedge j \neq 1\} \cup \{\mathrm{SP}_2\langle i\rangle \mid (s_1^{(i)}, n_2 + 1) \in \Delta_4\}$$

$F_{\mathrm{new}} = F_1 \cup F_2 \cup F_3 \cup F_4$.

Замечание 12 *Построение F_{new} с точностью до обозначений и группировки случаев C и D, а также E и F (для рассмотрения, соответственно, точек типа (3) и (4)), следует теореме 2 [14].*

Замечание 13 *Точки типа (1), согласно рассуждениям, следующим за теоремой 1 [14], появляются в случаях если полином, который обеспечил её вхождение в Δ_{new} (назовём его $f^{(i)}$) либо лежит в F_V либо может быть переведён в $\mathrm{VALPOL}(u^{(n')})$ с помощью теоремы 1 [14]. В первом случае соответствующая компонента вектора aux равна 0 (по построению) и формула $\mathrm{BP}\langle i, aux[i]\rangle = \mathrm{BP}\langle i, 0\rangle = f^{(i)}$ оставит полином неизменным, во втором случае эта формула в точности реализует теорему 1 [14].*

Замечание 14 *Существование k из построения полинома для точек типа (2) упоминается непосредственно в теореме 2 [14] со ссылкой на доказательство леммы 8.*

Шаг 6. Построить G_{new}, исходя из условий: $G_{\mathrm{new}} \subset G \cup F_N$ и $|G_{\mathrm{new}}| = |F_{\mathrm{new}}| - 1$. Пронумеровать элементы G_{new}, следуя следующему правилу. Каждый полином $g \in G_{\mathrm{new}}$ получает номер $i \in [1, l-1]_{\mathbb{N}}$, если выполнено условие:

$$s_k^{(i+k-1)} = r_k - t_k + 1, \tag{6}$$

где $k \in \{1, 2\}$, $\boldsymbol{t} = \mathrm{LP}(g)$ и

$$\boldsymbol{r} = \begin{cases} \boldsymbol{p}^{(i)} \in PG, & \text{если } g \in G; \\ \boldsymbol{n}, & \text{если } g \in F_N. \end{cases} \tag{7}$$

Замечание 15 *Условие $G_{new} \subset G \cup F_N$ указано в [14, п. 5, шаг 2]. Условие $|G_{new}| = |F_{new}| - 1$ взято из определения G. Формула (6) взята из [14, (12)], с учетом, что для $g \in F_N$ (элементов G, добавившихся на текущей итерации) в роли $\boldsymbol{p}^{(i)}$ выступает мультиномер текущей итерации — $\boldsymbol{n}$.*

Шаг 7. Построить множества T_{new}, PG_{new}, DG_{new} для G_{new}, следуя описанию T, PG и DG. Поясним способ построения PG_{new}: если $g^{(i)}$ из G_{new} принадлежит G, то его параметр $\boldsymbol{p}^{(i)}$ просто берется из PG, в противном случае, если он принадлежит F_N, то его параметр $\boldsymbol{p}^{(i)}$ полагается равным $\boldsymbol{n}$.

Заменить F, G, T, PG, DG на $F_{\text{new}}, G_{\text{new}}, T_{\text{new}}, PG_{\text{new}}, DG_{\text{new}}$.

Шаг 8. $\boldsymbol{n} := \boldsymbol{n}'$;

if $\boldsymbol{n} = \boldsymbol{p}$ **then** exit; **else goto** Шаг 1.

Замечание 16 *Условие окончания или продолжения алгоритма совпадает с указанным в [14, п. 5, шаг 3].*

1.4 Пример выполнения BMS-алгоритма

Рассмотрим следующую последовательность $u : \Sigma_{\mathbf{0}}^{(5,1)} \to \mathbb{F}_2 = \{0, 1\}$ [14]:

$$\begin{aligned} &u_{(0,0)} = 0, \quad u_{(0,1)} = 1, \quad u_{(0,2)} = 0, \\ &u_{(1,0)} = 1, \quad u_{(1,1)} = 1, \\ &u_{(2,0)} = 0, \\ &u_{(3,0)} = 0. \end{aligned}$$

Применим к ней BMS-алгоритм, используя схему, построенную в предыдущем разделе. Назовём «итерацией алгоритма» один проход по шагам 1–8 и будем нумеровать итерации римскими цифрами. Номера шагов в соответствии со схемой будут обозначаться арабскими цифрами.

Некоторые шаги данной итерации, на которых делать ничего не потребуется, будем пропускать, не оговаривая это каждый раз особо. Например, если после шага 1 $F_N = \emptyset$, то надо сразу переходить на шаг 8 (по goto из шага 3).

Пусть заданы начальные значения из шага 0: $\boldsymbol{n} := (0,0)$, $F := \{f^{(1)} = 1\}$, $(\Delta(F) = \emptyset)$, $G := DG := PG := \emptyset$.

I. Шаг 1. $\boldsymbol{s}^{(1)} = (0,0) \leqslant \boldsymbol{n} = (0,0)$.
$f^{(1)}[u]_{(0,0)} = f_{(0,0)}u_{(0,0)} = 1 \cdot 0 = 0 \Longrightarrow f^{(1)} \in F_V$. $F_N = \emptyset$.

Условие шага 3 выполнено тривиально ($F_N = \emptyset$). Перейдём на шаг 8.

Шаг 8. $\boldsymbol{n} := (0,1)$.

II. Шаг 1. $\boldsymbol{s}(1) = (0,0) \leqslant \boldsymbol{n} = (1,0)$
$f^{(1)}[u]_{(1,0)} = f_{(0,0)}u_{(1,0)} = 1 \Longrightarrow f^{(1)} \in F_N$.

Шаг 2 Вычислим $aux[1]$:

$$\Delta = \emptyset \Rightarrow \boldsymbol{n} \notin \mathrm{LP}(f^{(1)}) + \Delta(= \emptyset) \Rightarrow \mathrm{inSD}(\boldsymbol{s}^{(1)}) = 0 \Rightarrow aux[1] = 0.$$

Условие шага 3 не выполнено. Переходим на шаг 4.

Шаг 4. $\Delta_1 = \Delta_2 = \emptyset$, $\Delta_3 = \{(1,0)\}$ $(i = j = 1)$, $\Delta_3 = \{(1,0)\}$ $(i = j = 1)$.

Шаг 5. $F_3 = \{x_1^{n_1 - s_1^{(1)}+1} f^{(1)} = x_1^2\}$, $F_4 = \{x_2^{n_2 - s_2^{(1)}+1} f^{(1)} = x_2\}$.

Шаг 6. $G_{\text{new}} = g^{(1)} = 1$.

Шаг 7. $PG_{\text{new}} = \{\boldsymbol{p}^{(1)} = (1,0)\}$, $DG_{\text{new}} = \{d^{(1)} = 1\}$. $F = \{f^{(1)} = x_1^2; f^{(2)} = x_2\}$, $\Delta = \{(0,0); (1,0)\}$.

Шаг 8. $\boldsymbol{n} := (0,1)$.

III. Шаг 1. $\boldsymbol{s}^{(1)} = (2,0) \nleqslant \boldsymbol{n} = (0,1) \Longrightarrow f^{(1)} \in F_V$.
$\boldsymbol{s}^{(2)} = (0,1) \leqslant \boldsymbol{n} = (0,1)$.
$f^{(2)}[u]_{(0,1)} = f_{(0,1)}u_{(0,1)} = 1 \cdot 1 = 1 \Longrightarrow f^{(2)} \in F_N$.

Шаг 2. $\boldsymbol{n} = (0,1) \in \boldsymbol{s}^{(2)} + \Delta = \{(0,1); (1,1)\} \Longrightarrow aux[2] = 1$.

Шаг 3. Условие выполнено. $f^{(2)} := \mathrm{BP}\langle 2,1\rangle = x_2 + x_1$.

Шаг 8. $\boldsymbol{n} := (2,0)$ $(F = \{f^{(1)} = x_1^2; f^{(2)} = x_2 + x_1\})$.

IV. Шаг 1. $\boldsymbol{s}^{(1)} = (2,0) \leqslant \boldsymbol{n} = (2,0)$.
$f^{(1)}[u]_{(2,0)} = f_{(2,0)}u_{(2,0)} = 1 \cdot 0 = 0 \Longrightarrow f^{(2)} \in F_V$.
$\boldsymbol{s}^{(2)} = (0,1) \not\leqslant \boldsymbol{n} = (2,0) \Longrightarrow f^{(2)} \in F_V$.

Шаг 8. $\boldsymbol{n} := (1,1)$.

V. Шаг 1. $\boldsymbol{s}^{(1)} = (2,0) \not\leqslant \boldsymbol{n} = (1,1) \Longrightarrow f^{(1)} \in F_V$.
$\boldsymbol{s}^{(2)} = (0,1) \leqslant \boldsymbol{n} = (1,1)$.

$$f^{(2)}[u]^{(1,1)} = \sum f_{\boldsymbol{m}} u_{\boldsymbol{m}+(1,0)} = f_{(1,0)}u_{(2,0)} + f_{(0,1)}u_{(1,1)} = \\ = 0 + 1 = 1 \Longrightarrow f^{(2)} \in F_N.$$

Шаг 2. $\boldsymbol{n} = (1,1) \in \boldsymbol{s}^{(2)} + \Delta = \{(0,1);(1,1)\}$, $aux[2] = 1$.

Шаг 3. Условие выполнено. $f^{(2)} := \mathrm{BP}\langle 2,1\rangle = x_2 + x_1 + 1$. Переход на шаг 8.

Шаг 8. $ptn := (0,2)$ $(F = \{f^{(1)} = x_1^2; f^{(2)} = x_2 + x_1 + 1\})$.

VI. Шаг 1. $\boldsymbol{s}^{(1)} = (2,0) \not\leqslant \boldsymbol{n} = (0,2) \Longrightarrow f^{(1)} \in F_V$.
$\boldsymbol{s}^{(2)} = (0,1) \leqslant \boldsymbol{n} = (0,2)$.

$$f^{(2)}[u]_{(0,2)} = \sum f_{\boldsymbol{m}} u_{\boldsymbol{m}+(0,1)} = \\ = f_{(0,0)}u_{(0,1)} + f_{(1,0)}u_{(1,1)} + f_{(0,1)}u_{(0,2)} = \\ = 1 + 1 + 0 = 0 \Longrightarrow f^{(2)} \in F_V.$$

Шаг 8. $\boldsymbol{n} := (3,0)$.

VII. Шаг 1. $\boldsymbol{s}^{(1)} = (2,0) \leqslant \boldsymbol{n} = (3,0)$.
$f^{(1)}[u]_{(3,0)} = \sum f_{\boldsymbol{m}} u_{\boldsymbol{m}+(1,0)} = f_{(2,0)}u_{(3,0)} = 0 \Longrightarrow f^{(1)} \in F_V$.
$\boldsymbol{s}^{(2)} = (0,1) \not\leqslant \boldsymbol{n} = (3,0) \Longrightarrow f^{(2)} \in F_V$.

Шаг 8. $\boldsymbol{n} := (2,1)$.

2 Реализация

Реализация алгоритма выполнена с использованием языка программирования общего назначения C++ в соответствии с действующим стандартом [18], и в совокупности с широким распространением компиляторов с этого языка может использоваться на большом числе программно-аппаратных платформ. Подготовлены бинарные сборки для платформы x86-GNU/Linux (бинарный формат ELF).

При реализации использованы две библиотеки программных кодов с открытыми исходными кодами:

- NTL, версия 5.4.2 [20] — содержит реализацию арифметики в конечных полях;
- Boost, версия 1.37.0 [21] — использованы средства, повышающие удобство использования STL.

Кроме того, NTL может быть опционально скомпилирована с библиотекой GMP (GNU Multiple Precision Arithmetic Library [22]) для повышения производительности, что и было сделано в данном случае (версия GMP 4.2.2). Использованные в работе библиотеки либо проверены временем — разработка NTL относится к началу 90-х и с тех пор стабильно поддерживается, — либо имеют мощное сообщество пользователей и разработчиков, куда входят, в частности, профессионалы, ответственные за развитие языка C++ (Boost).

Реализация спроектирована в объектно-ориентированном стиле с широким применением STL, которая вносит элементы аппликативного программирования. Работу по реализации можно разделить на три части:

1. Арифметический процессор: полиномы от двух переменных, двумерные последовательности, точки дискретной плоскости (с учётом дифференциации точек, накладываемой алгоритмом).
2. Определение контейнерных структур данных для представления совокупностей полиномов, которыми в разных контекстах оперирует алгоритм.

3. Реализация алгоритма.

Ниже охарактеризованы основные особенности каждой из этих подсистем.

2.1 Арифметический процессор

Стоит отметить, что NTL содержит реализацию арифметики в кольцах полиномов лишь от одной переменной; самостоятельных библиотек с открытой реализацией полиномов двух (многих) переменных аналогичного NTL и Boost класса найти не удалось. Был создан класс полиномов от двух переменных, а также реализованы необходимые операции с объектами этого класса.

Реализация последовательно использует механизм шаблонов C++. В первую очередь это относится к параметризации типов коэффициентов полинома и элементов последовательности. Такое решение обусловлено несколькими причинами. Одна из них — дизайн NTL, в которой конечные поля различных типов (простые, расширенные, характеристики 2) представлены совокупностью не связанных между собой какими-либо языковыми средствами классов. В такой ситуации обеспечение полиморфизма создаваемого кода не может осуществляться при помощи традиционных для объектно-ориентированного программирования механизмов наследования и виртуальных функций. С другой стороны, шаблоны хорошо показывают себя при работе с набором классов, реализующих некоторый «неявный интерфейс» (случай NTL), обеспечивая параметрический полиморфизм [23].

Удобство использования шаблонов C++ совместно с классами NTL отчасти может быть обусловлено тем, что библиотека во многих моментах полагается на механизм макросов (это остаётся, однако, практически не заметным для её пользователя): шаблоны создавались, в том числе, как замена небезопасным в использовании макросам и, решая аналогичные задачи, шаблоны — быть может, против воли создателей — достаточно хорошо сочетаются с макросами.

Вторая причина попытки создать код, абстрагированный от типов конкретных алгебраических структур, состоит в том, что исходный алгоритм

Берлекэмпа—Месси находит всё новые интерпретация, расширения и обобщения. В том числе, рассматривались варианты алгоритма над различными типами колец и модулями [12, 13], кроме того, в более поздних, чем основная [14] для данной реализации, работах Сакаты изучалась задача построения группы полиномов с предписанными нулями в предположении, что нули лежат в некотором расширении поля коэффициентов многочленов [25]. В этой ситуации одной из задач стало создание максимально гибкой реализации основных модулей программы для облегчения дальнейшего изучения алгоритма и его приложений.

Как и в случае классического алгоритма Берлекэмпа—Месси, в расчётах не используется умножение двух полиномов общего вида, а — только умножение полинома на моном, что делает разумным отдельную реализацию этой операции, которая работает более эффективно, чем умножение в общем случае.

Из других особенностей реализации подсистемы арифметического процессора можно указать решение, связанное с операцией $f[u]_{\boldsymbol{n}}$. Здесь использован паттерн проектирования, известный под именем Прокси-класс (или Заместитель) [24]. Одной из задач реализации была удобочитаемость кода, в обеспечении которой большую роль играет перегрузка операций C++. В данном случае была перегружена операция обращения по индексу (subscript operator) класса полинома. Однако, эта операция не может быть сделана тернарной (что позволило бы легко приблизить её вызов к математической записи). В таких случаях можно воспользоваться дополнительным классом (обычно его называют прокси-класс), который сохраняет информацию, переданную бинарной операции обращения по индексу, то есть «запоминает» ссылки на полином и последовательность, и для которого определена перегруженная операция вызова функции (call operator), принимающая объект-точку дискретной плоскости. Последняя выполняет необходимые вычисления, а в коде это выглядит так: `f[u](n)`. Важным является тот факт, что прокси-класс является деталью реализации, которая остаётся максимально скрытой для клиента класса полинома.

2.2 Контейнерные структуры данных

В ходе выполнения алгоритма конструируются и изменяются ряд множеств. Было выделено три типа данных:

- `PolySet` для хранения текущего минимального множества.
- `PolyWithAuxInfo` для хранения «вспомогательного» (auxiliary) множества G.
- `PolySubset` для хранения подмножеств минимального множества F_V и F_N.

Первые два основаны на стандартном типе множества C++ `std::set` и определяют несколько дополнительных операций, используемых в алгоритме. Выбор `std::set` обусловлен необходимостью поддерживать указанные множества в порядке, таком что старшие степени образуют гиперболический набор: здесь используется особенность типа множества C++, которое в действительности является *упорядоченным* множеством. Пример добавленной операций доставляет добавленная в `PolySet` проверка того, что данная точка находится в сдвинутом в данную точку Δ-множестве, определяемом данным минимальным множеством полиномов (т. е. объектом `PolySet`). Кроме того, в обоих типах присутствует операция обращения по индексу для того, чтобы код, реализующий процедуру Берлекэмпа, наиболее выразительно отражал математическую запись.

`PolyWithAuxInfo` является множеством, хранящим тройки: полином, точка дискретной плоскости и элемент поля (упакованные в структуру C++), — сочетая в себе, таким образом, три множества из описания алгоритма: G, PG, DG. Множества старших степеней для F и G, то есть S и T, также не хранятся отдельно: степень является полем класса полинома, и потому информация из S и T содержится в объектах `PolySet` и `PolyWithAuxInfo` соответственно.

Третий контейнерный тип `PolySubset` реализован как стандартный список (`std::list`) структур, содержащих пары: итератор, указывающий на полином в текущем минимальном множестве, и индекс этого полинома.

Таким образом, для хранения полиномов в подмножествах исключено дополнительное копирование. Это можно считать вариантом паттерна Приспособленец (Flyweight) [24]. Здесь сыграли роль не только соображения эффективности, но и особенности алгоритма: рассматривая полином в подмножестве нужно иметь возможность модифицировать его либо удалять — так, чтобы эти изменения отразились на самом минимальном множестве.

Стоит отметить, что простое (отдельно стоящее) определение псевдонима типа (`typedef`) списка структур для случая `PolySubset` выполнить нельзя, потому что тип итератора по множеству полиномов зависит от шаблонных параметров полинома, тогда возможное определение псевдонима было бы шаблонным, что запрещено текущим стандартом C++. Эта сложность известная под названием template typedefs будет снята в следующем стандарте C++, где определения «шаблонных псевдонимов» будут разрешены [19].

2.3 Реализация алгоритма

Весь алгоритм был оформлен в виде отдельного класса. Объектно-ориентированный подход к построению реализации алгоритма позволил произвести достаточно глубокую декомпозицию, чтобы отразить основные шаги алгоритма. Были выделены основные фазы алгоритма, код выполнения которых помещался в отдельные методы. Определены, во-первых, данные, используемые и последовательно изменяемые несколькими фазами, а значит, подлежавшие оформлению в виде полей класса, и, во-вторых, данные, являющиеся локальными для каждого шага или даже его части. Таким образом, принятые в ходе проектирования решения помогают глубже понять природу и особенности алгоритма.

Известно, что объектно-ориентированный подход позволяет последовательно строить сколько угодно сложные абстракции, равномерно распределяя общую сложность между различными уровнями абстракции. Два описанных выше программных модуля содержали более низкоуровневые объекты и операции. Реализация же самого алгоритма в сравнении с ними

представляется более высокоуровневой. Поясним этот тезис.

Класс алгоритма `Algorithm` содержит единственный публичный метод `computeMinimalSet` для получения минимального множества по заданной в конструкторе последовательности, и приведённая в разделе 1.3 схема алгоритма практически дословно отображается на тело цикла этого метода:

```
buildFNandFV();      // шаг 1
if (FN.empty()) {  // goto на шаге 3
    continue;
}
// в условии ниже: шаг 2 и if из шага 3
// параметр degreeInvariantSuppliers — множество аих
if (isAtTheDegreeInvariantPoint(degreeInvariantSuppliers)) {
    renewF(degreeInvariantSuppliers); // then из шага 3
} else {
    buildNewDeltaSet();                    // шаг 4
    PolySet<T,S> F_new = buildNewF(); // шаг 5
    buildNewG(F_new);                      // шаг 6
    F = F_new;
}
```

Таким образом, за самым верхним уровнем абстракции, когда мы просто получаем решение (клиентский код, вызывающий `computeMinimalSet`), сразу идёт уровень, решающий задачу в терминах схемы алгоритма (тело `computeMinimalSet`). Далее, как видно из приведённого кода, сложность распределена по методам класса `Algorithm`. Общее количество этих методов около пятнадцати штук, но основные из них уже видны выше. Большая часть этих методов имеет довольно лаконичное тело, так как они, в свою очередь, полагаются на более низкие уровни абстракции, описанные, в том числе, в разделах 2.1 и 2.2.

Исходные коды реализации приведены в приложении.

3 Применение BMS-алгоритма для декодирования АГ-кодов типа кодов Рида—Соломона

3.1 АГ-коды типа кодов Рида—Соломона

Рассмотрим конструкцию семейства кодов [17], которые иногда называют алгебро-геометрическими кодами типа кодов Рида—Соломона. Пусть задано конечное поле $\mathbb{F}_q$ из q элементов, полином $C(x, y) \in \mathbb{F}_q[x, y]$. Множество точек (x, y), координаты x и y которых лежат в алгебраическом замыкании $\mathbb{F}$ поля $\mathbb{F}_q$, таких что $C(x, y) = 0$ называется *афинной кривой*, а сами точки — точками афинной кривой $C(x, y)$. Точка афинной кривой называется рациональной, если обе её координаты принадлежат $\mathbb{F}_q$.

Для удобства расчёта параметров строящегося семейства кодов на кривую накладывают дополнительные ограничения. *Полной степенью полинома* f от любого конечного числа s переменных $\{x_i\}_{i=1}^s$ назовём максимальную сумму степеней переменных в терме (члене) f:

$$f = \sum_{\alpha \in \mathbb{N}_0^s} f_\alpha x^\alpha : \deg f \stackrel{\text{def}}{=} \max_{f_\alpha \neq 0} \sum_i \alpha_i.$$

Определение корректно, так как лишь конечное число f_α отличны от нуля.

Для полинома $f \in \mathbb{F}_q[x_1, \ldots, x_s]$ определим операцию *гомогенизации* [26] $f \mapsto f^h$, $f^h \in \mathbb{F}_q[x_0, \ldots, x_s]$:

$$f^h(x_0, x_1, \ldots, x_s) \stackrel{\text{def}}{=} x_0^{\deg f} f\left(\frac{x_1}{x_0}, \ldots, \frac{x_s}{x_0}\right).$$

Таким образом, число переменных полинома увеличивается на одну, и он становится *однородным*, то есть суммарные степени переменных в каждом терме совпадают (и равны $\deg f$).

Важное свойство однородных полиномов состоит в том, что для любого корня $P \in \mathbb{F}_q^s$ однородного полинома, λP, где $\lambda \in \mathbb{F}_q$, также является

его корнем. Далее, говоря о корне однородного полинома, будет подразумеваться корень с точностью до множителя λ.

Можно определить вложение $\mathbb{F}^s \hookrightarrow \mathbb{F}^{s+1}$, при котором все корни исходного полинома f станут корнями f^h (нужно положить новую координату равной 1, оставив неизменными старые). Конечно, у f^h могут появиться и другие корни. Используя геометрический язык, говорят, что кривая f^h является *проективным замыканием* кривой f [26].

Определим *формальную частную производную* полинома $f \in \mathbb{F}_q[x_1, \ldots, x_s]$ по переменной x_i: $f \mapsto f_{x_i}$, $f_{x_i} \in \mathbb{F}_q[x_1, \ldots, x_s]$,

$$f = \sum_{\alpha \in \mathbb{N}_0^s} f_\alpha x^\alpha : \; f_{x_i} \stackrel{\text{def}}{=} \sum_{\substack{f_\alpha \neq 0 \\ \alpha_i > 0}} \alpha_i f_\alpha x^{\alpha - e_i},$$

где запись $\alpha_i f_\alpha$ означает $\sum_{k=1}^{\alpha_i} f_\alpha$, а вектор $e_i \in \mathbb{N}_0^s$ имеет все нулевые координаты, за исключением i-ой, которая равна 1.

Точка P кривой f называется *особой* (*сингулярной*), если все формальные частные производные в ней равны нулю. В противном случае точка называется *неособой* (*гладкой*). Кривая называется *регулярной*, если её проективное замыкание не имеет особых точек.

Ограничение, о котором упоминалось выше, состоит в том, что для построения рассматриваемого семейства кодов используются регулярные кривые $C(x, y)$.

Будем считать, что на парах целых неотрицательных чисел $\Sigma_{\mathbf{0}} = \mathbb{N}_0^2$ (а значит, на множестве мономов из $\mathbb{F}_q[x, y]$) введён линейный порядок $<_{\mathrm{T}}$ (см. раздел 1.2). Пусть $C(x, y) = 0$ — уравнение регулярной кривой степени m (то есть старшая относительно $<_{\mathrm{T}}$ степень многочлена $C(x, y)$ равна m), а $\{P_i = (x_i, y_i)\}_{i=1}^n$ — рациональные точки на ней. Выберем целое j, такое, что

$$m - 2 \leqslant j \leqslant \left\lfloor \frac{n-1}{m} \right\rfloor.$$

Мономы из множества $\{x^a y^b \mid (a, b) \leqslant_{\mathrm{T}} (0, j)\}$ перенумеруем так: $\{\varphi_i\}_{i=0}^{\mu}$, где $\mu = \mu(j)$ — более точно, эта нумерация определяется биективным отоб-

ражением

$$(a, b) \mapsto ((a + b)^2 + a + 3b)/2. \tag{8}$$

Код типа Рида—Соломона $C^*(j)$ задаётся проверочной матрицей $\mathbf{H}$:

$$\mathbf{H} = \begin{bmatrix} \varphi_0(P_1) & \dots & \varphi_0(P_n) \\ \varphi_1(P_1) & \dots & \varphi_1(P_n) \\ \vdots & & \vdots \\ \varphi_\mu(P_1) & \dots & \varphi_\mu(P_n) \end{bmatrix} \tag{9}$$

Пусть k это размерность данного кода, а d_{min} — его минимальное расстояние. Можно доказать [16], что:

$$k = n - (mj - g + 1),$$

$$d_{min} \geq d^* = mj - 2g + 2,$$

где g это параметр, называемый *родом кривой*. В случае регулярной кривой степени m

$$g = (m - 1)(m - 2)/2.$$

Величина d^* называется *конструктивным кодовым расстоянием*.

3.2 Метод декодирования АГ-кодов типа кодов Рида—Соломона Юстесена—Ларсена—Йенсена—Хохольда

Ниже будет изложен метод декодирования, первоначально описанный в [17].

Пусть по каналу пришло слово $\mathbf{r} \in \mathbb{F}_q^n$, и $\mathbf{r} = \mathbf{c} + \mathbf{e}$, где $\mathbf{c}$ это кодовое слово, а $\mathbf{e}$ — вектор ошибок, возникших при передаче. Задача декодера — определить вектор ошибок $\mathbf{e}$ [17, раздел IV].

Синдром $\mathbf{s} \in \mathbb{F}_q^\mu$ определяется так:

$$\mathbf{s} = \mathbf{H}\mathbf{r}^\tau = \mathbf{H}\mathbf{e}^\tau.$$

Предположим, что ошибки произошла в позициях, отвечающих точкам (x_i, y_i), $i \in I \subset \{1, \ldots, n\}$. Обозначим

$$S_{ab} = \sum_{i \in I} e_i x_i^a y_i^b, \quad (a, b) \leqslant_{\mathrm{T}} (0, j). \tag{10}$$

Неизвестными являются, как позиции ошибок (x_i, y_i), так и их величины e_i. Как и в классическом декодере Питерсона—Горенстейна—Цирлера [15] для кодов Рида—Соломона, эти две проблемы решаются отдельно. Первая связана с введением *полинома локаторов ошибок*:

$$\sigma(x, y) = \sum_{l,k} \sigma_{lk} x^l y^k,$$

который по определению содержит среди своих корней $\{P_i\}_{i \in I}$ и не делится на $C(x, y)$.

Полином локаторов ошибок обладает одним важным свойством. Рассмотрим выражение $\sum_{l,k} \sigma_{lk} S_{a+l,b+k}$:

$$\begin{aligned} \sum_{l,k} \sigma_{lk} S_{a+l,b+k} &= \sum_{l,k} \sigma_{lk} \sum_{i \in I} e_i x_i^{a+l} y_i^{b+k} = \\ &= \sum_{i \in I} e_i x_i^a y_i^b \sum_{l+k \leqslant h} \sigma_{lk} x_l^a y_i^k = 0. \end{aligned} \tag{11}$$

Здесь a и b берутся такими, что $(a+l, b+k) \leqslant_{\mathrm{T}} (0, j)$, то есть $S_{a+l,b+k}$ имеет смысл. Последнее равенство выполняется, потому что внутренняя сумма содержит полином σ, вычисленный в своих корнях $P_i = (x_i, y_i)$.

Соотношения (11) должны быть поданы на вход BMS-алгоритму. Многочлен минимальной степени из минимального множества, полученного по BMS-алгоритму, будет многочленом локаторов ошибок, если количество ошибок в канале t удовлетворяет неравенству [17, Теорема 4]:

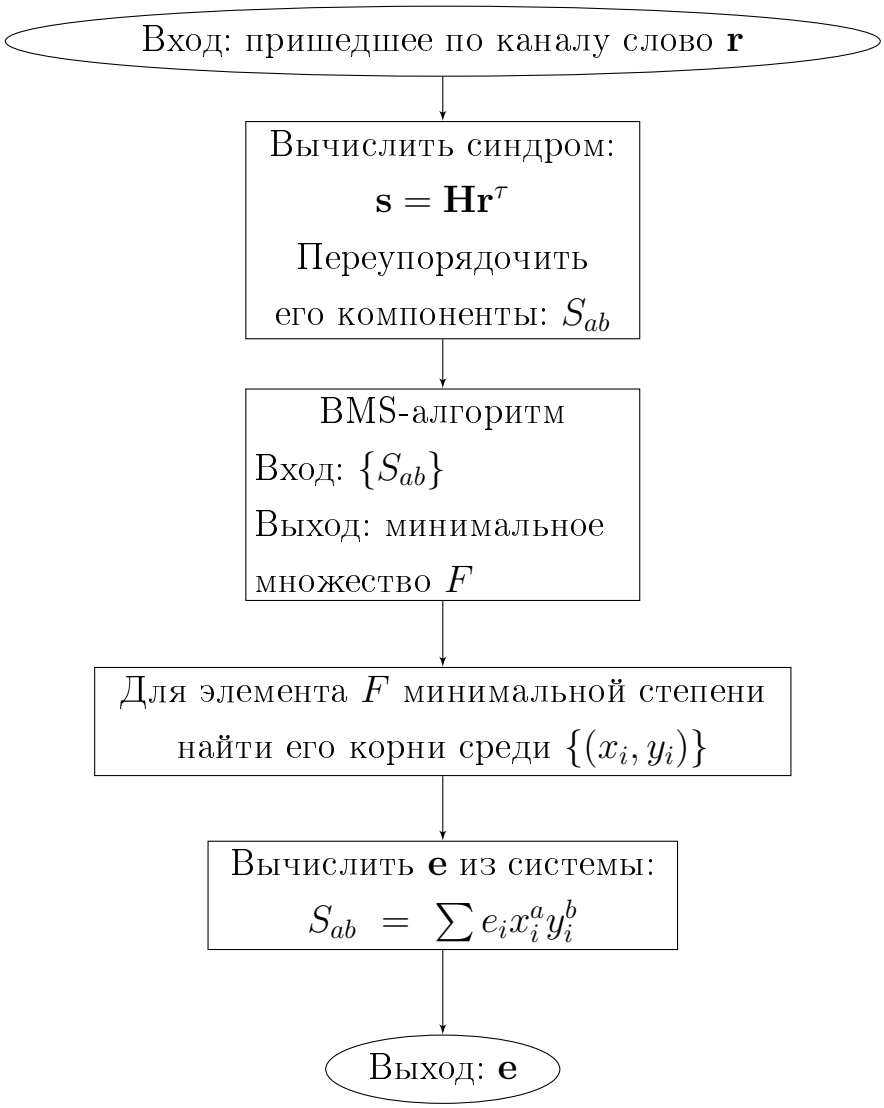
$$t < d^*/2 - m^2/8 + m/4 - 1/8. \tag{12}$$

После нахождения σ, нужно определить его корни из числа $\{P_i\}$, под-

ставить их в (10) и решить её относительно вектора ошибок $\mathbf{e}$ одним из общих методов решения систем линейных уравнений.

В заключение сделаем замечание о границе для числа исправляемых ошибок (12). Она более ограничительна, чем привычное для классических (не алгебро-геометрических) помехоусточивых кодов, $t < d^*/2$. Оказывается, в случае АГ-кодов также можно исправлять ошибки до половины конструктивного кодового расстояния d^*. Это было доказано Фенгом и Рао [27].Применение их техники «голосования большинством» (majority voting) к рассматриваемым кодам можно найти в [28]. Основная идея состоит в нахождении дополнительных синдромов (для получения большего числа уравнений (11)), используя уравнение кривой и информацию из BMS-алгоритма на промежуточных шагах. Как ясно, это требует более тонкого встраивания BMS-алгоритма в схему декодирования.

3.3 Схема алгоритма декодирования



Переупорядочение в первом блоке производится с помощью отображения (8).

Заключение

В данной работе построена схема BMS-алгоритма, которая может служить руководством для независимых реализаций. Выполнена реализация BMS-алгоритма на языке программирования C++. Рассмотрена возможность применения алгоритма для декодирования алгебро-геометрических кодов типа кодов Рида—Соломона.

Дальнейшая работа может быть проведена по нескольким направлениям. Интерес представляет реализация построенной в разделе 3.3 схемы декодера. Кроме того, как отмечено в разделе 3.2, схема может быть усложнена для повышения числа исправляемых декодером ошибок за счёт более сложного встраивания BMS-алгоритма в декодер.

Имеются работы, обсуждающие n-мерную версию BMS-алгоритма [25] (мы рассмотрели двумерную версию), которая также может стать предметом дальнейшей разработки и реализации. Использование её при декодировании кодов на кривых высших размерностей также может быть проведено, однако трудность практического применения такого кодека заключается в отсутствии явных конструкций многомерных кривых с достаточно богатым запасом рациональных точек [17].

Список литературы

[1] Шеннон К. Математическая теория связи. – В сб. «Работы по теории информации и кибернетике». М., Иностранная литература, 1963.

[2] Goppa V. D., Codes associated with divisors // Prohl. Peredach, Inform. 1977. Vol. 13. No. 1. Pp. 33-39,

[3] Tsfasman M. A., Vladut S. G., Zink T., Modular curves, Shimura curves and Goppa codes, better than Varshamov—Gilbert bound // Math. Nuchr. 1982. Vol. 104. Pp. 13–28.

[4] Алфёров А. П., Зубов А. Ю., Кузьмин А. С., Черёмушкин А. В. Основы криптографии: учебное пособие. — М.: Гелиос АРВ. 2005.

[5] O'Sullivan M. New codes for the Berlekamp-Massey-Sakata algorithm // Finite Fields Appl. 2001. No 7. Pp. 293–317.

[6] Пеленицын А.М. О реализации декодера одного класса алгебро-геометрических кодов с использованием алгоритма Сакаты // Неделя науки 2008: сб. тезисов. Том 1. — Ростов н/Д: Изд-во ЮФУ. 2008. С. 55–57.

[7] Маевский А.Э., Пеленицын А.М. О программной реализации алгебро-геометрического кодека с применением алгоритма Сакаты // В сб. «Материалы X Международной научно-практической конференции „Информационная безопасность“», ч.2. Таганрог. ЮФУ. 2008. С. 55–57.

[8] Маевский А.Э., Пеленицын А.М. Реализация программного алгебро-геометрического кодека с применением алгоритма Сакаты // Изв. ЮФУ. Технические науки. 2008. №8. С. 196–198.

[9] Berlekamp E. R. Algebraic Coding Theory – New York: McGrow Hill, 1968. (Перевод: Берлекэмп Э. Алгебраическая теория кодирования. – М.: Мир, 1971.)

[10] Massey J.L., Shift Register Synthesis and BCH Decoding, // IEEE Trans. Inform. Theory. 1969, Vol. IT-15. No. 1.

[11] Gashkov S.B., Gashkov I.B. The Berlekamp-Massey Algorithm. A Sight from Theory of Pade Approximants and Orthogonal Polynomials // LNCS. 2004. Vol. 3037. Pp. 561–564.

[12] Kurakin V.L., Kuzmin A.S., Mikhalev A.V., Nechaev A.A. Linear recurring sequences over rings and modules // I. of Math. Science. Contemporary Math. and it's Appl. Thematic surveys. 1994. Vol. 10. I. of Math. Sciences. 1995. Vol. 76. № 6.

[13] Куракин В.Л. Алгоритм Берлекэмпа—Месси над коммутативными артиновыми кольцами главных идеалов // Фундаментальная и прикладная математика. 1999. Том 5. Вып. 4.

[14] Sakata S. Finding a minimal set of linear recurring relations capable of generating a given finite two–dimensional array // J. Symb. Comp. 1988. Vol. 5. Pp. 321–337.

[15] Блейхут Р. Теория и практика кодов, контролирующих ошибки — М.: Мир, 1986.

[16] Justesen J., Larsen K.J., Havemose A., Jensen H.E., and Høholdt T. Construction and Decoding of a Class of Algebraic Geometry Codes // IEEE Trans. Inform. Theory. 1989. Vol. 35, Pp. 811–821.

[17] Justesen J., Larsen K.J., Jensen H. E., and Høholdt T. Fast decoding of codes from algebraic plane curves // IEEE Trans. Inform. Theory. 1992. Vol. 38. Pp. 111-119.

[18] ISO Information Technology — Programming Languages — C++ Document Number ISO/IEC 14882-1998 ISO/IEC, 1998.

[19] Working Draft, Standard for Programming Language C++, 2009-03-23 (Черновик стандарта C++).

[20] NTL: A Library for doing Number Theory by Victor Shoup, веб-сайт: http://shoup.net/ntl/

[21] Boost: C++ Libraries, веб-сайт: http://www.boost.org/

[22] GNU Multiple Precision Arithmetic Library, веб-сайт: http://gmplib.org/

[23] Вандевурд Д., Джосаттис Н. Шаблоны C++: справочник разработчика. — М.: Издательский дом «Вильямс», 2003.

[24] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2001.

[25] Sakata S. N-dimensional Berlekamp-Massey algorithm for multiple arrays and construction of multivariate polynomials with preassigned zeros // LNCS. Vol. 357. 1989. Pp. 356–376.

[26] Кокс Д., Литтл Дж., О'Ши Д. Идеалы, многообразия и алгоритмы. Введение в вычислительные аспекты алгебраической геометрии и коммутативной алгебры. — М.: Мир, 2000.

[27] Feng G. L., Rao T. R. N., Decoding algebraic-geometric codes up to the designed minimum distance // IEEE Trans. Inform. Theory. 1993. Vol. 39. Pp. 37–45.

[28] Sakata S., Justesen J., Madelung Y., Jensen H. E., Høholdt T., Fast decoding of AG-codes up to the designed minimum distance // IEEE Trans. Inform. Theory. 1993. Vol. 41. Pp. 1672–1677.

Приложение

Большинство классов являются шаблонными, и, в соответствии с правилами C++, должны целиком помещаться в заголовочные файлы. Однако для большего удобства объявления и определения объёмных шаблонных классов разнесены в разные файлы. При таком разделении создавался файл, содержащий всего две директивы `#include`, включавшие текст файла с объявлением и текст файла с определением членов шаблонного класса — такие вспомогательные файлы не включены в листинг.

BMSAlgorithm.h

```
#ifndef _BMSALGORITHM_H
#define _BMSALGORITHM_H

#include <algorithm>
#include <cassert>
#include <iostream>
#include <map>
#include <set>
#include <utility>
#include <vector>

#include "PolySet.h"
#include "Polynomial.h"
#include "Sequence.h"
#include "DeltaSet.h"

namespace bmsa {

using std::vector;
using std::pair;
using namespace boost::lambda;
using namespace boost::tuples;

template<typename T, typename S = T> // T = Poly Coef Type
                                     // S = Seq Elem Type
```

```
class Algorithm {
    typedef std::list<Point> PointCollection;
    typedef std::list<PolySubsetEntry<T,S> > PolySubset;

    Sequence<S> const & u;

    Point n; // current step of the algorithm

    PolySet<T,S> F; // current minimal set
    PolyWithAuxInfoSet<T,S> G;

    PolySubset FN;
    PolySubset FV;

    vector<typename Polynomial<T,S>::Coef> discr; // current
                                                  // discrepancies
    vector<size_t> degreeInvariantSuppliers;
    DeltaSet newDeltaSet;

public:
    Algorithm(Sequence<S> const & u_) : u(u_),
            F(ElemTypeTraits<typename Polynomial<T,S>::Coef>::id()
               ) {}

    PolySet<T,S> computeMinimalSet();

private:
    void clearParams();

    void buildFNandFV();

    bool isAtTheDegreeInvariantPoint(
            vector<size_t> & degreeInvariantSuppliers);

    void renewF(vector<size_t> const & degreeInvariantSuppliers);

    // Berlekamp procedure
    Polynomial<T,S> BP(size_t i, size_t j);
```

```
    // Subsidiary procedure -- the confluent version of BP
    template <size_t K>
    Polynomial<T,S> SP(size_t);

    void buildNewDeltaSet();
    PolySet<T,S> buildNewF();
    void buildNewG(PolySet<T,S> const &);
    bool fnBetweenFs(Polynomial<T,S> const & f,
                     Polynomial<T,S> const & fNext,
                     Polynomial<T,S> const & fn) const;

    // constitutes building new delta-set
    void addPointsOfTypeI();
    void addPointsOfTypeII();
    void addPointsOfTypeIII();
    void addPointsOfTypeIV();

    DeltaSetPoint
    pointOfTypeIFromPolySubsetEntry(PolySubsetEntry<T,S> const &);

    Polynomial<T,S>
    buildPolyFromDeltaSetPoint(DeltaSetPoint const &);
};

} // namespace bmsa
#endif  /* _BMSALGORITHM_H */
```

BMSAlgorithm.cc

```
#include <algorithm>
#include <stdexcept>
#include <iterator>
#include <iostream>

#include <cstddef> // size_t

#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>
#include <boost/function.hpp>
```

```
#include "Polynomial.h"
#include "PolySet.h"
#include "BMSAlgorithm.h"

using std::cout;
using std::endl;

namespace bmsa {

template<typename T, typename S>
PolySet<T,S> Algorithm<T,S>::computeMinimalSet() {
    for (n = Point::nil; totalLess(n, u.size()); ++n ) {
        clearParams();
        buildFNandFV(); // find polynomials non-valid
                        // at the current step
        if (FN.empty()) {
            cout << "F_N is empty - F would not change" << endl;
            continue;
        }
        if (// n is in the degree-invariant set for each f in FN
                isAtTheDegreeInvariantPoint(
                    degreeInvariantSuppliers)) {
            cout << "Delta-set would not change" << endl;
            // delta-set and size of F unchanged
            renewF(degreeInvariantSuppliers);
            std::cout << "F renewed:\n\t" << F << std::endl;
        } else {
            buildNewDeltaSet();
            cout << "New delta-set: "<< endl << "\t" <<
                newDeltaSet;
            PolySet<T,S> F_new = buildNewF();
            buildNewG(F_new);
            F = F_new;
        }
    }
    return F;
}
```

```
template<typename T, typename S>
void Algorithm<T,S>::clearParams() {
    FN.clear();
    FV.clear();
    degreeInvariantSuppliers.clear();
    degreeInvariantSuppliers.resize(F.size());
    discr.clear();
    newDeltaSet.clear();
}

template<typename T, typename S>
void Algorithm<T,S>::buildFNandFV() {
    size_t i = 0;
    for(
            typename PolySet<T,S>::const_iterator fIt = F.begin();
            fIt != F.end();
            ++fIt, ++i) {
        typename Polynomial<T,S>::Coef d; // current discrepancy;
        // for f in F: if f[u](n) could be evaluated...
        // .. and discrepancy not null then
        // f is non-valid (goes to FN)
        if ( fIt->getDegree() <= n
                && (d = (*fIt)[u](n)) != typename Polynomial<T,S
                    >::Coef() ) {
            discr.push_back(d);
            FN.push_back(PolySubsetEntry<T,S>(i, fIt));
        }
        else {
            discr.push_back(typename Polynomial<T,S>::Coef());
            FV.push_back(PolySubsetEntry<T,S>(i, fIt));
        }
    }
}

// test wether n is degree-invariant point for all f in FN;
// if yes, degreeInvariantSuppliers holds indices of aux poly's
// which help to preserve the degree in the Berlekamp-procedure;
// this info should be used in such a way (pseudocode):
//      for f in FN:
```

```
//            f = BP<f, degreeInvariantSuppliers[indexOf(f)]>
template<typename T, typename S>
bool Algorithm<T,S>::isAtTheDegreeInvariantPoint(
        vector<size_t> & degreeInvariantSuppliers) {
    bool result = true;
    for(typename PolySubset::const_iterator it = FN.begin();
            it != FN.end(); ++it) {
        size_t degInv
                = F.isInShiftedDeltaSet(n, it->polyIt->getDegree()
                    );
        result = !(degInv == PolySet<T,S>::
           NOT_IN_SHIFTED_DELTA_SET);
        degreeInvariantSuppliers[it->polyIndex] = degInv;
    }
    return result;
}

template<typename T, typename S>
void Algorithm<T,S>::renewF(vector<size_t> const &
   degreeInvariantSuppliers) {
    for(typename PolySubset::const_iterator it = FN.begin();
            it != FN.end(); ++it) {
        size_t nonvalidPolyIdx = it->polyIndex;
        F.replace(it->polyIt, BP(nonvalidPolyIdx,
                degreeInvariantSuppliers[nonvalidPolyIdx]));
    }
}

template<typename T, typename S>
void Algorithm<T,S>::buildNewDeltaSet() {
    addPointsOfTypeI();
    addPointsOfTypeII();
    addPointsOfTypeIII();
    addPointsOfTypeIV();
}

// though polys from FV give points of type I,
// they are not added to delta-set
// (due to some technical reasons arise in building F_new
```

```
// from this delts-set and possible improoved efficiency);
// buildNewF() (builds F_new) take this into account;
template<typename T, typename S>
void Algorithm<T,S>::addPointsOfTypeI() {
    for (typename PolySubset::const_iterator it = FN.begin(); it
       != FN.end(); ++it)
        if (degreeInvariantSuppliers[it->polyIndex]
                != PolySet<T,S>::NOT_IN_SHIFTED_DELTA_SET)
            newDeltaSet.push_back(
                    DeltaSetPoint(DeltaSetPoint::I, it->polyIndex,
                            degreeInvariantSuppliers[it->polyIndex
                               ]));
}

template<typename T, typename S>
DeltaSetPoint
Algorithm<T,S>::pointOfTypeIFromPolySubsetEntry(PolySubsetEntry<T,
   S> const & psentry) {
    return DeltaSetPoint(DeltaSetPoint::I, psentry.polyIndex,
            degreeInvariantSuppliers[psentry.polyIndex]);
}

template<typename T, typename S>
void Algorithm<T,S>::addPointsOfTypeII() {
    typename PolySubset::iterator it = std::adjacent_find(FN.begin
       (), FN.end(),
            adjacentPolyIndiciesForPolySubset<T,S>);
    while (it != FN.end()) {
        newDeltaSet.push_back(
            DeltaSetPoint(DeltaSetPoint::II, it->polyIndex,
                it->polyIndex)
        );
        it = std::adjacent_find(++it, FN.end(),
                adjacentPolyIndiciesForPolySubset<T,S>);
    }
}

template<typename T, typename S>
void Algorithm<T,S>::addPointsOfTypeIII() {
```

```
    using namespace boost::lambda;
    for (typename PolySubset::const_iterator jIt = FN.begin(); jIt
        != FN.end(); ++jIt) {
        const Point s_j = jIt->polyIt->getDegree();
        for (typename PolySet<T,S>::reverse_iterator kIt = F.
            rbegin(); kIt != F.rend(); ++kIt)
            if ( n[1] >= s_j[1] + kIt->getDegree()[1]) {
                if (n[0] >= s_j[0] + kIt->getDegree()[0]) {
                    typename PolySubset::const_iterator iIt
                        = std::find_if(FN.begin(), FN.end(),
                            bind(
                                polyItWithPolySubsetEntryComparator<
                                T,S>, --kIt.base(), _1));
                    if (iIt != FN.end())
                        newDeltaSet.push_back(DeltaSetPoint(
                            DeltaSetPoint::III,
                                iIt->polyIndex, jIt->polyIndex));
                    else
                        break;
                }
                else
                    break;
            }
    }
}

template<typename T, typename S>
void Algorithm<T,S>::addPointsOfTypeIV() {
    using namespace boost::lambda;
    for (typename PolySubset::const_iterator jIt = FN.begin(); jIt
        != FN.end(); ++jIt) {
        const Point s_j = jIt->polyIt->getDegree();
        for (typename PolySet<T,S>::const_iterator kIt = F.begin()
            ; kIt != F.end(); ++kIt)
            if ( n[0] >= s_j[0] + kIt->getDegree()[0]) {
                if (n[1] >= s_j[1] + kIt->getDegree()[1]) {
                    typename PolySubset::const_iterator iIt
                        = std::find_if(FN.begin(), FN.end(),
```

```
                        bind(
                            polyItWithPolySubsetEntryComparator<
                            T,S>, kIt, _1));
                    if (iIt != FN.end())
                        newDeltaSet.push_back(DeltaSetPoint(
                            DeltaSetPoint::IV,
                                iIt->polyIndex, jIt->polyIndex));
                    else
                        break;
                }
                else
                    break;
            }
    }
}

template<typename T, typename S>
Polynomial<T,S> Algorithm<T,S>::buildPolyFromDeltaSetPoint(
    DeltaSetPoint const& pt) {
    std::cout << "DSPoint: " << pt;
    Polynomial<T,S> res;
    size_t k;
    Point t;
    switch(pt.pointType) {
        case DeltaSetPoint::I:
            res = BP(pt.i, pt.j); break;
        case DeltaSetPoint::II:
            t = Point(n[0] - F[pt.i].getDegree()[0] + 1,
                    n[1] - F[pt.j].getDegree()[1] + 1);
            k = findPolyWithLessDegIdxInPolySubset<T,S>(FN, t);
            res = BP(k, pt.i); break;
        case DeltaSetPoint::III:
            if (pt.i != F.size() - 1)
                res = BP(pt.j, pt.i);
            else
                res = SP<0>(pt.j);
            break;
        case DeltaSetPoint::IV:
            if (pt.j != 0)
```

```
                res = BP(pt.i, pt.j - 1);
            else
                res = SP<1>(pt.i);
            break;
        default: std::logic_error("Illegal type of point in delta-
            set!");
    }
    return res;
}

template<typename T, typename S>
PolySet<T,S> Algorithm<T,S>::buildNewF() {
    using namespace boost::lambda;
    PolySet<T,S> res;
    add(res, FV); // as points from FV weren't add to newDeltaSet
    for(DeltaSet::const_iterator it = newDeltaSet.begin();
            it != newDeltaSet.end(); ++it) {
        Polynomial<T,S> p = buildPolyFromDeltaSetPoint(*it);
        res.insert(p);
    }
    return res;
}

template<typename T, typename S>
bool gBetweenFs(Polynomial<T,S> const & f, Polynomial<T,S> const &
    fNext,
        PolyWithAuxInfo<T,S> const & g) {
    return f.getDegree()[0] == g.p[0] - g.poly.getDegree()[0] + 1
        && fNext.getDegree()[1] == g.p[1] - g.poly.getDegree()[1]
            + 1;
}

template<typename T, typename S>
bool Algorithm<T,S>::fnBetweenFs(
        Polynomial<T,S> const & f,
        Polynomial<T,S> const & fNext,
        Polynomial<T,S> const & fn) const {
    return f.getDegree()[0] == n[0] - fn.getDegree()[0] + 1
        && fNext.getDegree()[1] == n[1] - fn.getDegree()[1] + 1;
```

```
}

template<typename T, typename S>
void Algorithm<T,S>::buildNewG(PolySet<T,S> const & F_new) {
    PolyWithAuxInfoSet<T,S> GNew;
    for (typename PolySet<T,S>::const_iterator fit = F_new.begin()
        ,
             fitNext = ++F_new.begin();
             fit != --F_new.end(); ++fit, ++fitNext) {
        bool gfound = false;
        for (typename PolyWithAuxInfoSet<T,S>::iterator git = G.
            begin();
                 git != G.end() && !gfound;
                 ++git) {
            if (gBetweenFs(*fit, *fitNext, *git)) {
                GNew.insert(*git);
                G.erase(git); // git will not be used any more
                gfound = true;
            }
        }
        if (gfound)
            continue;
        for(typename PolySubset::iterator fnit = FN.begin();
                 fnit != FN.end() && !gfound;
                 ++fnit) {
            Polynomial<T,S> const & fnPoly = *(fnit->polyIt);
            if (fnBetweenFs(*fit, *fitNext, fnPoly)) {
                GNew.insert( PolyWithAuxInfo<T,S>(fnPoly, n,
                    fnPoly[u](n)) );
                FN.erase(fnit);
                gfound = true;
            }
        }
        G = GNew;
    }
}

// Berlekamp procedure
template<typename T, typename S>
```

```
Polynomial<T,S> Algorithm<T,S>::BP(size_t i, size_t j) {
    assert( 0 <= i && i < F.size());
    assert( 0 <= j && j < G.size());

    std::cout << "BP<" << i << ", " << j << ">" << endl;
    Polynomial<T,S> const & f = F[i];
    Polynomial<T,S> const & g = G[j].poly;
    Point r = Point(
        std::max(f.getDegree()[0], g.getDegree()[0] + n[0] - G[j].
            p[0]),
        std::max(f.getDegree()[1], g.getDegree()[1] + n[1] - G[j].
            p[1])
    );
    Polynomial<T,S> f1 = f.onMonomialMultiply(r - f.getDegree());
    Polynomial<T,S> f2 = g.onMonomialMultiply(r - n + G[j].p - g.
        getDegree());
    typename Polynomial<T,S>::Coef c = (-discr[i] / G[j].d);
    Polynomial<T,S> res = f1 + c * f2;
    return res;
}

template<typename T, typename S>
template<size_t K>
inline
Polynomial<T,S> Algorithm<T,S>::SP(size_t m) {
    Point deg;
    deg[K] = n[K] - F[m].getDegree()[K] + 1;
    return F[m].onMonomialMultiply(deg);
}

} // namespace bmsa
```

PolySet.h

```
#ifndef _POLYSET_H
#define _POLYSET_H

#include <algorithm>
#include <iostream>
```

```
#include <iterator>
#include <limits>
#include <list>
#include <set>
#include <stdexcept>
#include <utility>

#include <boost/lambda/bind.hpp>

#include "Polynomial.h"

namespace bmsa {

template<typename T, typename S = T>
class PolySet {
    typedef Polynomial<T,S> PolynomialT;
    typedef std::set<PolynomialT> Container;

    Container data;

public:
    typedef typename Container::iterator iterator;
    typedef typename Container::const_iterator const_iterator;
    typedef typename Container::reverse_iterator reverse_iterator;
    typedef typename Container::const_reference const_reference;
    typedef typename Container::reference reference;

    static size_t NOT_IN_SHIFTED_DELTA_SET;

    PolySet() {}

    PolySet(PolynomialT const & p) {
        data.insert(p);
    }

    const_iterator begin() const        {return data.begin();}

    const_iterator end() const          {return data.end();}
```

```
    reverse_iterator rbegin()           {return data.rbegin();}

    reverse_iterator rend()             {return data.rend();}

    std::pair<iterator, bool>
    insert(PolynomialT const & poly)
        {return data.insert(poly);}

    iterator insert(iterator pos, PolynomialT const & poly)
        {return data.insert(pos, poly);}

    typename Container::size_type size() const
        {return data.size();}

    size_t isInShiftedDeltaSet(Point const & pt, Point const &
       shift);

    void replace(const_iterator const & oldEntry,
            PolynomialT const & newEntry);

    // 0-based poly index in PolySet
    PolynomialT const & operator[](int n) const;

};

template<typename T, typename S>
inline
typename PolySet<T,S>::PolynomialT const &
PolySet<T,S>::operator[](int n) const {
    const_iterator it = data.begin();
    std::advance(it, n);
    return *it;
}

template<typename T, typename S>
inline
void PolySet<T, S>::replace(const_iterator const & oldEntry,
        Polynomial<T, S> const & newEntry) {
    std::cout << "Replacing a poly in the PolySet:" << std::endl
```

```
        << "\told entry: " << *oldEntry << std::endl;
    data.erase(oldEntry);
    std::cout << "\tnew entry: " << newEntry << std::endl;
    data.insert(newEntry);
}

template<typename T, typename S>
inline
std::ostream& operator<<(std::ostream& os, PolySet<T, S> const &
   ps) {
    os << "{ ";
    std::copy(ps.begin(), ps.end(),
            std::ostream_iterator<Polynomial<T, S> >(os, "; "));
    os << "}";
    return os;
}
/*************************************************************/

template<typename T, typename S>
struct PolySubsetEntry {
    size_t polyIndex;
    typename PolySet<T,S>::const_iterator polyIt;
    //Polynomial::Coef discrepancy;

    // constructor from fields
    PolySubsetEntry(
            size_t polyIndex_,
            typename PolySet<T,S>::const_iterator const & polyIt_)
                :
                polyIndex(polyIndex_), polyIt(polyIt_) {}
};

template<typename T, typename S>
inline
bool adjacentPolyIndiciesForPolySubset(
        PolySubsetEntry<T,S> const & pse1,
        PolySubsetEntry<T,S> const & pse2) {
    return pse1.polyIndex + 1 == pse2.polyIndex;
}
```

```
template<typename T, typename S>
inline
bool polyItWithPolySubsetEntryComparator(
        typename PolySet<T,S>::iterator it,
        PolySubsetEntry<T,S> const & pse) {
    return it == pse.polyIt;
}

template<typename T, typename S>
inline
bool greaterDegreeThenInPolySubsetEntry(Point const & deg,
        PolySubsetEntry<T,S> const & pse) {
    return pse.polyIt->getDegree() < deg;
}

template<typename T, typename S, typename PolySubset>
inline
size_t findPolyWithLessDegIdxInPolySubset(PolySubset const & ss,
   Point t) {
    using namespace boost::lambda;
    typename PolySubset::const_iterator it = std::find_if(
            ss.begin(),
            ss.end(),
            bind(greaterDegreeThenInPolySubsetEntry<T,S>, t, _1));
    if (it == ss.end())
        throw std::logic_error("No poly in given subset which"
                " degree is less then given");
    return it->polyIndex;
}

template<typename T, typename S>
inline
std::ostream& operator<<(std::ostream& os, PolySubsetEntry<T,S>
   const & pse) {
    os << *pse.polyIt << " idx: " << pse.polyIndex;
    return os;
}
/*************************************************************/
```

```
template<typename T, typename S>
struct PolyWithAuxInfo {
    Polynomial<T,S> poly;

    Point p; // 'potential' -- last step at which corresponding
             // polynomial had been valid for given sequence;
    typename Polynomial<T,S>::Coef d; //discrepancy

    PolyWithAuxInfo(Polynomial<T,S> const & poly_, Point const &
       p_,
            typename Polynomial<T,S>::Coef const & d_)
            : poly(poly_), p(p_), d(d_) {}

    Polynomial<T,S>      const * getPoly()     const   {return &poly
       ;}
    Point                const * getP()        const   {return &p;}
};

template<typename T, typename S>
inline
std::ostream& operator<<(std::ostream& os, PolyWithAuxInfo<T,S>
   const & pwai) {
    os << pwai.poly << " pot: " << pwai.p << " discr: " << pwai.d;
    return os;
}

template<typename T, typename S>
inline
bool operator<(PolyWithAuxInfo<T,S> const & pwai1,
               PolyWithAuxInfo<T,S> const & pwai2) {
    return pwai1.p[0] - pwai1.poly.getDegree()[0]
           > pwai2.p[0] - pwai2.poly.getDegree()[0]
           && pwai1.p[1] - pwai1.poly.getDegree()[1]
                < pwai2.p[1] - pwai2.poly.getDegree()[1];
}

template<typename T, typename S>
class PolyWithAuxInfoSet {
```

```
    typedef std::set<PolyWithAuxInfo<T,S> > Container;

    Container data;

public:
    typedef typename Container::iterator iterator;
    typedef typename Container::const_iterator const_iterator;

    const_iterator begin() const        {return data.begin();}

    iterator begin()                    {return data.begin();}

    const_iterator end() const          {return data.end();}

    iterator end()                      {return data.end();}

    std::pair<iterator, bool>
    insert(PolyWithAuxInfo<T,S> const & pwai)
        {return data.insert(pwai);}

    void erase(iterator pos)            {return data.erase(pos);}

    typename Container::size_type size() const
        {return data.size();}

    PolyWithAuxInfo<T,S> const & operator[](size_t n) const;
};

template<typename T, typename S>
inline
PolyWithAuxInfo<T,S> const & PolyWithAuxInfoSet<T,S>::operator[](
   size_t n) const {
    const_iterator it = data.begin();
    std::advance(it, n);
    return *it;
}

template<typename T, typename S>
inline
```

```
std::ostream& operator<<(std::ostream& os,
        PolyWithAuxInfoSet<T,S> const & pwaiset) {
    std::copy(pwaiset.begin(), pwaiset.end(),
            std::ostream_iterator<PolyWithAuxInfo<T,S> >(os, "; ")
               );
    return os;
}


template<typename T, typename S>
size_t PolySet<T,S>::NOT_IN_SHIFTED_DELTA_SET
                              = std::numeric_limits<size_t>::max();

// returns the 0-based index of F-element for which
// pt is in shifted delta-set;
// NOT_IN_SHIFTED_DELTA_SET if n is not in shifted delta-set
// for any f from F
template<typename T, typename S>
size_t PolySet<T,S>::isInShiftedDeltaSet(Point const & pt, Point
   const & shift) {
    if (data.size() < 2)
        return NOT_IN_SHIFTED_DELTA_SET; // delta-set is empty
    size_t fIdx = 0;
    for(const_iterator fIt = data.begin(); fIt != --data.end();
            ++fIt, ++fIdx) {
        const_iterator fNextIt(fIt);
        ++fNextIt;
        if (pt[0] < shift[0] + fIt->getDegree()[0]
            && pt[1] < shift[1] + fNextIt->getDegree()[1])
            return fIdx;
    }
    return NOT_IN_SHIFTED_DELTA_SET;
}

template<typename T, typename S>
Polynomial<T,S> polyFromPolySubsetEntry(PolySubsetEntry<T,S> const
    & pse) {
    return *pse.polyIt;
}
```

```
template<typename T, typename S, typename PolySubset>
void add(PolySet<T,S> & ps, PolySubset const & pss) {
    std::transform(pss.begin(), pss.end(), std::inserter(ps, ps.
        begin()),
            polyFromPolySubsetEntry<T,S>);
}

} // namespace bmsa
```

Poly.h

```
#ifndef _POLY_H
#define _POLY_H
#include <algorithm>
#include <deque>
#include <iostream>
#include <vector>

#include "ElementType.h"
#include "Sequence.h"
#include "Point.h"

namespace bmsa {

template <typename CoefT, typename SeqElemT = CoefT>
class Polynomial {
public:
    typedef CoefT Coef;
    typedef std::deque<Coef> OneDimStorage;
    typedef std::deque< OneDimStorage > StorageT;

    typedef Point DegreeT;

private:
    class ProxyPolySeqProduct {
        Polynomial const & poly;
        Sequence<SeqElemT> const & seq;
    public:
```

```
        ProxyPolySeqProduct(Polynomial const & poly_,
                Sequence<SeqElemT> const & seq_) : poly(poly_),
                   seq(seq_) {}

        typename Polynomial::Coef operator()(Point const & pr)
           const;
    };

    friend class ProxyPolySeqProduct;

public:
    Polynomial() {}

    Polynomial(Coef const & c) : coefs(1) {
        coefs[0].push_back(c);
    }

    StorageT const & getCoefs() const { return coefs; }

    void setCoefs(StorageT const & data);

    DegreeT getDegree() const { return degree; }

    Polynomial& operator*=(Coef const &);

    Polynomial onMonomialMultiply(DegreeT const & monomialDegree)
       const;

    Polynomial& operator+=(Polynomial const & rhs);

    ProxyPolySeqProduct operator[](Sequence<SeqElemT> const & seq)
        const;

private:
    StorageT coefs;

    DegreeT degree;

    void countDegree();
```

```
};
/*************** End of Polynomial ********************/

template<typename T, typename S>
inline
Polynomial<T, S> operator*(typename Polynomial<T,S>::Coef const &
    coef,
        Polynomial<T, S> poly) {
    return poly *= coef;
}

template<typename T, typename S>
inline
Polynomial<T, S> operator*(Polynomial<T, S> poly,
        typename Polynomial<T, S>::Coef const & coef) {
    return coef*poly;
}

template<typename T, typename S>
inline
Polynomial<T, S> operator+(Polynomial<T, S> lhs,
        Polynomial<T, S> const & rhs) {
    return lhs += rhs;
}

template<typename T, typename S>
inline
// for establishing delta-set
bool operator<(Polynomial<T, S> const & lhs, Polynomial<T, S>
    const & rhs) {
    return lhs.getDegree()[0] > rhs.getDegree()[0]
            && lhs.getDegree()[1] < rhs.getDegree()[1];
}

// format: [[a b c][e f]] - witout spaces in "]["
template<typename T, typename S>
std::istream& operator>>(std::istream& is, Polynomial<T, S> & poly
    );
```

```
template<typename T, typename S>
std::ostream& operator<<(std::ostream& os, Polynomial<T, S> const
    & poly);

} // namespace bmsa

#endif  /* _POLY_H */
```

Poly.cc

```
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
#include <ios>
#include <iterator>
#include <numeric>
#include <sstream>
#include <string>
#include <vector>

#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>
#include <boost/lambda/algorithm.hpp>

#include "GenericTwoDimIO.h"
#include "Poly.h"

namespace bmsa {

/****************** Arithmetic operations **********************/
template<typename T, typename S>
class OneDimAddition {
    typedef int IntType;
    typedef typename Polynomial<T,S>::OneDimStorage OneDimStorage;
public:
    OneDimStorage operator()(
            OneDimStorage const & lhs,
```

```
            OneDimStorage const & rhs) {
        if (lhs.size() < rhs.size() )
            return operator()(rhs, lhs);
        OneDimStorage res(lhs);
        std::transform(rhs.begin(), rhs.end(), lhs.begin(),
                res.begin(),
                std::plus<typename Polynomial<T,S>::Coef>());
        return res;
    }
};

template <typename T, typename S>
Polynomial<T,S>& Polynomial<T,S>::operator+=(Polynomial<T,S> const
    & rhs) {
    int lengthInFirstDimDif = coefs.size() - rhs.coefs.size();
    if (lengthInFirstDimDif < 0) {
        typename Polynomial<T,S>::StorageT::const_iterator it(rhs.
           coefs.begin());
        std::advance(it, coefs.size());
        std::copy(it, rhs.coefs.end(), std::back_inserter(coefs));
        std::transform(
                rhs.coefs.begin(), it,
                coefs.begin(),
                coefs.begin(),
                OneDimAddition<T,S>()
        );
    } else {
        std::transform(
                rhs.coefs.begin(), rhs.coefs.end(),
                coefs.begin(),
                coefs.begin(),
                OneDimAddition<T,S>()
        );
    }
    this->countDegree();
    return *this;
}

template <typename T, typename S>
```

```
void oneDimOnScalarMultiply(typename Polynomial<T,S>::
    OneDimStorage & data,
        typename Polynomial<T,S>::Coef const & coef) {
    using namespace boost::lambda;
    std::for_each(data.begin(), data.end(), _1 *= coef);
}

template <typename T, typename S>
Polynomial<T,S>& Polynomial<T,S>::operator*=(
        typename Polynomial<T,S>::Coef const & coef) {
    using namespace boost::lambda;
    if (coef == typename Polynomial::Coef() ) {
        coefs.clear();
        countDegree();
    } else
        std::for_each(coefs.begin(), coefs.end(),
                bind(oneDimOnScalarMultiply<T,S>, _1, coef));
    return *this;
}

template <typename T, typename S>
void oneDimOnMonomialMultiply(typename Polynomial<T,S>::
    OneDimStorage & data,
        int monomialDegree) {
    data.insert(data.begin(), monomialDegree,
                typename Polynomial<T,S>::Coef());
}

template <typename T, typename S>
Polynomial<T,S> Polynomial<T,S>::onMonomialMultiply(
        typename Polynomial::DegreeT const & monomialDegree) const
            {
    using namespace boost::lambda;
    Polynomial res(*this);
    std::for_each(res.coefs.begin(), res.coefs.end(),
            bind(oneDimOnMonomialMultiply<T,S>, _1,monomialDegree
                [1]));
    res.coefs.insert(res.coefs.begin(), monomialDegree[0],
       OneDimStorage());
```

```
    res.degree += monomialDegree;
    return res;
}

template <typename T, typename S>
typename Polynomial<T,S>::Coef
Polynomial<T,S>::ProxyPolySeqProduct::operator()(
        Point const & pt) const {
    typename Polynomial::Coef res = typename Polynomial<T,S>::Coef
       ();
    for(size_t i = 0; i < poly.coefs.size(); ++i)
        for(size_t j = 0; j < poly.coefs[i].size(); ++j)
            res += poly.coefs[i][j]
                    * seq( Point(i, j) + pt - poly.getDegree() );
    return res;
}
/*************** End of Arithmetic operations ***************/

template <typename T, typename S>
typename Polynomial<T,S>::DegreeT
countDegree(typename Polynomial<T,S>::StorageT const & coefs);

template <typename T, typename S>
void Polynomial<T,S>::setCoefs(StorageT const & data) {
    coefs = data;
    countDegree();
}

template <typename T, typename S>
typename Polynomial<T,S>::ProxyPolySeqProduct Polynomial<T,S>::
   operator[](
        Sequence<S> const & seq) const {
    return typename Polynomial<T,S>::ProxyPolySeqProduct(*this,
       seq);
}

/********** Degree-computing related routines **********/
template <typename T, typename S>
void Polynomial<T,S>::countDegree() {
```

```
    degree = findLastInSeq(coefs);
}
/******* End of Degree-coumting related routines **********/

/****************** Streaming operations *****************/
// format: [[a b c][e f]]
template <typename T, typename S>
std::istream& operator>>(std::istream& is, Polynomial<T,S> & poly)
    {
    typename Polynomial<T,S>::StorageT data;
    if (fillStorage(is, data))
        poly.setCoefs(data);
    return is;
}

template <typename T, typename S>
std::ostream& operator<<(std::ostream& os, Polynomial<T,S> const &
    poly) {
    print(os, poly.getCoefs());
    os << " degree: " << poly.getDegree();
    return os;
}
/*************** End of Streaming operations ***************/

} // namespace bmsa
```

GenericTwoDimIO.h

```
#ifndef _GENERICTWODIMIO_H
#define _GENERICTWODIMIO_H

#include <iostream>
#include <ios>

namespace bmsa {

template <typename TwoDimStorage>
std::istream& fillStorage(std::istream& is, TwoDimStorage& st) {
    if (is.peek() == '[')
```

```
        is.get();
    else
        is.setstate(std::ios::failbit);
    while(is) {
        char nextCh = is.peek();
        if (nextCh == '[')
            is.get();
        else if (nextCh == ']') {
            break;
        }
        else {
            is.setstate(std::ios::failbit);
            break;
        }
        typename TwoDimStorage::value_type v;
        while(is.peek() != ']') {
            typename TwoDimStorage::value_type::value_type c;
            is >> c;
            v.push_back(c);
        }
        is.ignore(2, ']');
        st.push_back(v);
    }
    return is;
}

template <typename TwoDimStorage>
std::ostream& print(std::ostream& os, TwoDimStorage const & data)
   {
    os << '[';
    for(
            typename TwoDimStorage::const_iterator vit = data.
               begin();
            vit != data.end();
            ++vit
        ) {
        os << '[';
        for(
                typename TwoDimStorage::value_type::const_iterator
```

```
                    it = vit->begin();
                it != vit->end();
                ++it
            ) {
            os << *it;
            if ((it + 1) != vit->end())
                os << ' ';
        }
        os << ']';
    }
    os << ']';
    return os;
}
} // namespace bmsa
#endif  /* _GENERICTWODIMIO_H */
```

DeltaSet.h

```
#ifndef _DELTASET_H
#define _DELTASET_H

#include <iostream>
#include <iterator>
#include <list>

#include <cstddef>

namespace bmsa {

// delta set of excluded points -- based on some delta set
// which points and the current sequence segment
// define the excluded points of four types;
// '(un)changed' below -- comparatively with the points of
// delta set on which this DS of excluded points is based on;
// type I   : both coordinates remain unchanged
// type II  : both coordinates changed
// type III: first coordinate changed
// type IV  : second coordinate changed
struct DeltaSetPoint {
```

```
    enum {I = 1, II, III, IV}; // possible point types

    size_t pointType;

    size_t i;

    size_t j;

    DeltaSetPoint(size_t pointType_, size_t i_, size_t j_) :
            pointType(pointType_), i(i_), j(j_) {}
};

inline
bool
operator<(DeltaSetPoint const & p1, DeltaSetPoint const & p2) {
     return p1.pointType < p2.pointType;
}

inline
std::ostream& operator<<(std::ostream& os, DeltaSetPoint const &
   pt) {
    os << "type: " << pt.pointType
       << " i=" << pt.i
       << " j=" << pt.j;
    return os;
}

typedef std::list<DeltaSetPoint> DeltaSet;

inline
std::ostream& operator<<(std::ostream& os, DeltaSet const & ds) {
    std::copy(ds.begin(), ds.end(),
            std::ostream_iterator<DeltaSetPoint>(os, "\n\t"));
    os << std::endl;
    return os;
}
} // namespace bmsa
#endif  /* _DELTASET_H */
```

Sequence.h

```
#ifndef _SEQUENCE_H
#define _SEQUENCE_H

#include <iostream>
#include <vector>

#include "ElementType.h"
#include "GenericTwoDimIO.h"
#include "Point.h"

namespace bmsa {

template<typename ElemType = NTL::GF2>
class Sequence {
public:
    typedef ElemType ElemT;
    typedef std::vector< std::vector<ElemT> > StorageT;

    Sequence(std::istream& is) :
            DEFAULT_VAL(ElemTypeTraits<ElemT>::Null())
        {is >> *this;}

    Point size() const {return size_;}

    ElemT const & operator()(Point const &) const;
private:
    StorageT rep;

    Point size_;

    const ElemT DEFAULT_VAL;

    template <typename T>
    friend
    std::istream& operator>>(std::istream& is, Sequence<T>& seq);

    template <typename T>
```

```
    friend
    std::ostream&
    operator<<(std::ostream& os, Sequence<T> const & seq);
}; // class Sequence

template <typename T>
typename Sequence<T>::ElemT
const & Sequence<T>::operator()(Point const & pt) const {
    if (pt[0] > static_cast<int>(rep.size())
            || pt[1] > static_cast<int>(rep[pt[0]].size()))
        return DEFAULT_VAL;
    return rep[pt[0]][pt[1]];
}

template<typename T>
inline
std::istream& operator>>(std::istream& is, Sequence<T>& seq) {
    typename Sequence<T>::StorageT st;
    if (fillStorage(is, st))
        seq.rep = st;
    seq.size_ = ++findLastInSeq(seq.rep);
    return is;
}

template<typename T>
inline
std::ostream& operator<<(std::ostream& os,
                         Sequence<T> const & seq) {
    print(os, seq.rep);
    return os;
}

} // namespace bmsa

#endif  /* _SEQUENCE_H */
```

Point.h

```
#ifndef _POINT_H
```

```
#define _POINT_H

#include <algorithm>
#include <cassert>
#include <cstddef>
#include <iostream>
#include <numeric>
#include <vector>
#include <tr1/array>

#include <boost/lambda/lambda.hpp>

namespace bmsa {

class Point {
    static const size_t DIMENSION = 2;

    typedef std::tr1::array<int, DIMENSION> ComponetsStorageT;
    ComponetsStorageT data;

public:
    static Point nil;

    Point() {
        data[0] = 0;
        data[1] = 0;
    }

    Point(int x, int y) {
        data[0] = x;
        data[1] = y;
    }

    int weight() const {
        return std::accumulate(data.begin(), data.end(), 0);
    }

    //subscript operators
    ComponetsStorageT::reference
```

```
    operator[](ComponetsStorageT::size_type n) {
        assert( 0 <= n && n < DIMENSION);
        return data[n];
    }

    ComponetsStorageT::const_reference operator[](
            ComponetsStorageT::size_type n) const {
        assert( 0 <= n && n < DIMENSION);
        return data[n];
    }

    inline
    bool operator==(Point const & rhs) const {
        return (data == rhs.data);
    }

    Point& operator+=(Point const & rhs) {
        data[0] += rhs.data[0];
        data[1] += rhs.data[1];
        return *this;
    }

    Point& operator-=(Point const & rhs) {
        data[0] -= rhs.data[0];
        data[1] -= rhs.data[1];
        return *this;
    }

    Point& operator++();

    Point operator++(int);

    friend      //partial ordering
    bool byComponentLessOrEqual(Point const &, Point const &);

    friend
    std::ostream& operator<<(std::ostream& os, Point const &);
};
```

```
inline
bool operator!=(Point const & lhs, Point const & rhs) {
    return !(lhs == rhs);
}

// by component less
inline
bool operator<=(Point const & lhs, Point const & rhs) {
    return lhs[0] <= rhs[0] && lhs[1] <= rhs[1];
}

inline
bool operator<(Point const & lhs, Point const & rhs) {
    return (lhs <= rhs) && (lhs != rhs);
}

inline
bool totalLess(Point const & lhs, Point const & rhs) {
    int lhsPow = lhs.weight();
    int rhsPow = rhs.weight();
    if (lhsPow == rhsPow)
        return lhs[0] > rhs[0];
    else
        return lhsPow < rhsPow;
}

inline
bool totalLessOrEqual(Point const & lhs, Point const & rhs) {
    return lhs == rhs || totalLess(lhs, rhs);
}

inline
Point operator+(Point lhs, Point const & rhs) {
    // lhs: pass-by-copy optimization
    return lhs += rhs;
}

inline
Point operator-(Point lhs, Point const & rhs) {
```

```
    return lhs -= rhs;
}

template< typename OneDimStorage >
int lastInOneDim(OneDimStorage const & data) {
    using namespace boost::lambda;
    int deg = data.rend()
                - std::find_if(data.rbegin(), data.rend(),
                    _1 != typename OneDimStorage::value_type())
                - 1;
    return deg > -1 ? deg : 0;
}

template< typename TwoDimStorge >
Point findLastInSeq(TwoDimStorge storage) {
    if (storage.empty())
        return Point();
    typedef std::vector<Point> PtStorage;
    PtStorage points;
    int i = 0;
    for(
            typename TwoDimStorge::const_iterator vit = storage.
               begin();
            vit != storage.end();
            ++vit, ++i ) {
        int lessDimDegree = lastInOneDim(*vit);
        points.push_back(Point(i, lessDimDegree));
    }

    return
        //using total ordering on Point type:
        *(std::max_element(points.begin(), points.end(), totalLess
           ));
}

} // namespace bmsa
#endif  /* _POINT_H */
```

Point.cc

```
#include <functional>
#include <numeric>
#include <sstream>

#include <boost/lambda/lambda.hpp>

#include "Point.h"

namespace bmsa {

Point Point::nil = Point();

inline
bool
byComponentLessOrEqual(Point const & lhs, Point const & rhs) {
    using namespace boost::lambda;
    return
        std::inner_product(lhs.data.begin(), lhs.data.end(),
                rhs.data.begin(),
                true, // init value
                std::logical_and<bool>(),
                _1 <= _2);
}

std::ostream& operator<<(std::ostream& os, Point const & pt) {
        os << '(';
        std::ostringstream oss;
        copy(pt.data.begin(), pt.data.end(),
                std::ostream_iterator<int>(oss, ","));
        std::string s = oss.str();
        std::string ss(s.begin(), --(s.end()));
        os << ss << ')';
        return os;
}

Point& Point::operator++() {
    if (data[0] == 0) {
```

```
        data[0] = data[1] + 1;
        data[1] = 0;
    } else {
        --data[0];
        ++data[1];
    }
    return *this;
}

Point Point::operator++(int) {
    Point old(*this);
    ++*this;
    return old;
}

} // namespace bmsa
```

ElementType.h

```
#ifndef _ELEMENTTYPE_H
#define _ELEMENTTYPE_H

namespace bmsa {

template <typename T>
struct ElemTypeTraits {
    static T id() {
        T res = T();
        return ++res;
    }

    static T Null() {
        return T();
    }
};
} // namespace bmsa
#endif  /* _ELEMENTTYPE_H */
```