Отчёт о программе.

Федоров Глеб M33351

Январь 2021

Оглавление

1 Документация к коду

1.1 Представление полиномов

В данной программе полиномы от нескольких переменных хранятся в виде абстрактного синтаксического дерева (далее AST). Каждый узел данного дерева(объект класса Node) может быть константой, переменной, унарной или бинарной операцией. Для типа Node* существует синоним типа PolynomialTree. Поддерживается единственная унарная операция - унарный минус (не используется за ненадобностью).

Поддерживаются следующие бинарные операции:

1. **Sum - сумма двух мономов**
2. **Multiplication - произведение двух мономов**
3. **Exponentiation - возведение переменной x в степень n**

Узел дерева удовлетворяет следующей грамматике:

$$Node ::= Constant|Variable|SumNodeNode|MultiplicationNodeNode|ExponentiationVariableConstant$$

Будем считать что **моном** задаётся следующей грамматикой:

$$Monom ::= Constant|Variable|MultiplicationMonomMonom|ExponentiationVariableConstant$$

Будем называть **термом** выражение в следующей грамматике:

$$Term ::= Constant|Variable|ExponentiationVariableConstant$$

Для каждого объекта класса Node определены следующие методы:

1. **clone() - создание копии данного объекта**
2. **to_str() - строковое представление полинома**
3. **get_monomials() - получение списка мономов данного полинома**
4. **get_term() - получение списка термов данного полинома**

Опишем работу метода **to_str()**: так как операция $-$ реализована в виде домножения на -1, то метод **to_str()** выводит полином в виде $\sum_{i=1}^{n} \delta * x_i$, где $\delta = -1|\_$, а x_i - моном. Пример: $(x^2 - y).to\_str() = x^{2.000000} + -1.000000 * y$.

Опишем классы, которые не расширяют набор методов класса **Node**:

1. **Constant** - позволяет хранить константы. Каждая хранится как *longdouble*
2. **Variable** - позволяет хранить константы. Название каждой константы хранится как std::string

Класс **AbstractBinaryOperation** хранит схой левый и правый аргумент в виде **Node***, свою ассоциативность в виде **Associativity**, и своё строковое представление. Каждый наследник класса **AbstractBinaryOperation**, помимо методо, определённых в **Node** реализует следующие:

1. getLeftNode() - возвращает левый аргумет
2. getRightNode() - правый левый аргумет
3. set_left_node(Node*) - изменяет левый аргумент
4. set_right_node(Node*) - изменяет левый аргумент

Для объекта класса $Node*$ определена функци **get_LT(**$Node*$**)** которая возвращает **LT** от полинома.

1.2 Парсер

Парсин входных данных просходит методом рекурсивного спуска, который реализован в классе **Parser**. Данный класс содержит единственный публичный метод **parse(std::string)**, который возвращает представление полинома в виде AST или бросает одно из следующих исключений:

1. **DivisionByZero** - попытка делить на ноль
2. **LostOperand** - пропущен операнд для бинарной операции
3. **UnsupprtedOperation** - будет брощено в следующих ситуциях
   (a) попытка возвести число в степень.
   (b) попытка возвести что-то в степень, не являющуюся числом.
   (c) попытка возвести переменную в не натуральную степень.
   (d) введено деление полиномов. Деление полиномов в процессе парсинга запрещено для упрощения парсера. Данная программа поддерживает деление полиномов представленных в виде **Node***.
4. **WrongToken** - введён неопознанный токен.

1.3 Simplify, Equals, Join

Поддерживается структурное упрощение полиномов при помощи функции **get_simplified**. Данная функция упорядочивает полином согласно лексикографического упорядочения, упрощает каждый моном, а затем складывает равные, с точностью до константы, мономы. Пример: $get_simplified(-1*2*3*x*x*y+x*y) = -6*x^2*y+x*y$.

Функция **sumIfEquals**(Node*, Node*) возвращает сумму двух мономов, если они равны с точность до константы, или nullptr иначе. Функция **equals**(std::vector<Node*> &t1_terms*, std::vector<Node*> &t1_terms*) возвращает $true$, если нормализованные мономы равны, и $false$ иначе.

Функция **normalize**(std::vector<Node *>::const_iterator from, std::vector<Node *>::const_iterator to) возвращает нормализованный моном.

Каждая из этих функция прозводит декомпозицию ABS в std::vector<Node*>(раскладывает на мономы), или работает с уже декомпозированным деревом. Для того чтобы заново собрать дерево реализована функция **join**(..., char delimiter), которая возвращает моном, если $delimiter = "*"$, или возвращает полином, если $delimiter = "+"$.

Для того, чтобы понять, какому именно типу принадлежит конкретный узел используются механихзмы полиморфизма в C++, а именно происходят попытки привести данный объект к нужному типу при помощи $dynamic_cast<*>()$, который возвращает nullptr, если преобазование не удалось.

1.4 Арифметические операции

Реализованны следующие арифметические операции для работы с полиномами: (далее $args = (Node*left, Node*right, MonomialOrder*order, MonomialOrder* service\_plex\_order)$)

1. **sum(args)** - возвращает сумму двух полиномов упорядоченную упорядочением order. Алгоритм суммирования следующий - создаётся объект типа Sum(left, right), затем вызывается метод get_simplified для нового объекта, который упрощает его согласно $service\_plex\_order$, а затем упорядочивает результат согласно $order$.

2. **multiply_to_monomial(args)** - создаётся множество объектво типа Multiplication, которые затем упрощаются при помощи $get\_simplified$, а после собираются в новый полином при помощи метода join.

3. **divide_monomials(args)** - декомпозирует $left$ и $right$, проверяет, возможно ли поделить $left$ на $right$, и если возможно, возвращает результат деления, упорядоченный согласно $order$, в противном случае возвращает $nullptr$.

4. **divide(args)** - реализован алгоритм с лекции.

1.5 Мономиальное упорядочение

Для работы с полиномами от нескольких переменных реализованы мономильноные упорядочения. Для удобства работы с упорядочениями они были реализованы синтаксически похоже на то, как они реализованы в maple. Каждое упорядочение является наследником класса **MonomialOrder**, который предоставляет следующие методы

1. конструктор **MonomialOrder** - принимает вектор переменны, по которым будет происходить упорядочение. Заполняется std::map<string, size_t> - переменная, приоритет.
2. виртуальный метод **compare** - сравнивает два монома. Если первый меньше или равен второго, то возвращается false, иначе true. Реализован в наследниках.
3. **sort_monomial** - упорядочивает моном согласно std::map<string, size_t>.
4. **add_other_variable** - расширает упорядочение новой переменной, с наименьшим приоритетом
5. **get_variables** - возвращает список используемых переменных.

Реализованны следующие упорядочения:

1. **Lex** - не имеет ничего общего с лексикографическим упроядочение. Упорядочивает **только** согласно std::map<string, size_t>. Используется в служебных целях.
2. **Plex** - истинное лексикографическое упорядочение.
3. **Grlex** - градуированное лексикографическое упорядочение.

Данные классы просто реализуют метод **compare**. Реализация метода **compare** для каждого упорядочения примерно одинаковая: декомпозируем мономы, получаем необходимую информацию, на основании неё делаем сравнение.

1.6 FGLM

Реализация алгоритма FGLM, который преобразует базис Грёбнера нульмерного идеала к иному мономиальному упорядочению. Класс FGLM содержит три публичных метода:

1. конструктор **FGLM** - принимает базис Грёбнера нульмерного идеала, старое мономиальное упорядочение, новое мономиальное упорядочение и список используемых переменных. Проверка на то что данный список полиномов является базисом Грёбнера, и то, что образуемый ими нульмерный идеал - нульмерный не происходит. В первом случае поведение программы не определено, во втором - программа не завершится.

2. **transform** - перобразует базис к новому упорядочению.

3. **get_in_maple_dls** - возвращает строку, которая является скриптом на языке Maple, который проверяет корректность работы алгоритма. Из-за особенности метода **to_str()** данный скрипт не всегда будет работать корректно, возможно придётся немного поправить полиномы.

Алгоритм FGLM заключается в поиске линейной комбинации элементов из списка $MBasis$, который является списком пар $[monomial, NormalFrom(monomil, oldBasis)]$, а именно, если $v = NormalForm(monom)$ представима в виде линейной комбинации $sum_{i=1}^{n} \lambda_i * MBasis[i].second$, то полином $v + \sum_{i=1}^{n} \lambda[i] * MBasis[i].first$ принадлежит базису Грёбнера с новым мономиальным упорядочением.

Линейная комбинация ищется только в том случае, если текущий $monom$ не является произведением предыдущих.

$monom$ генерируется следующим образом: если линейно комбинации, описанной выше нет, или она тривиальная, то мы добавляем произведение $monom$ на каждую переменные, которые входят в базис. Затем мы берём из множества максимальный элемент, согласно новому упорядочнию.

Алгоритм продолжает работу до тех пор, пока множество не пусто.

Опишем методы, которые используются для реализации алгоритма:

1. (get_norma_form($monom$)) - возвращает остаток от деления $monom$ на старый базис Грёбнера.
2. **is_product($monom$)** - возвращает true, если $monom$ является произведением предыдущих мономов, и false - иначе.
3. get_linear_relation(normal_form, MBasis, relation) - возвращает true и записывает линейную комбинацию в relation, если существует не тривиальная линейная комбинация, и возвращает false иначе. Линейная комбинация ищется следующим образом:
   (a) пусть MBasis.size() = n, тогда для каждого i домножим MBasis[i].second на $freeVar$, где $freeVar$ = @_$(n - i)$ - свежая переменная.
   (b) составим систему уравнений на коэффициенты, относительно свежих переменных
   (c) решим её
   (d) если свежие переменные не равны нудю одновременно, то составим линейную комбинацию, инача прост овернём false.

Система линейных уравнение решается при помощи библиотеки $MTL4$.

2 Тесты

2.1 Google_Tests

часть функций протестированно unit - тестами. Тестирование FGLM unit тестами не проводилось.

2.2 Тесты для FGLM

были протестированны следующие мономиальные идеалы(далее: идеал, старое упорядочение, новое упорядочени, результат):

1. $\{z^2-1, y^2-1, -y*z+x\}$, $Plex(\{x,y,z\})$, $Grlex(\{x,y,z\})$, $\{z^2-1, y*z-x, y^2-1, x*z-y, x*y-z, x^2-1\}$
2. $\{z-1, y-1, x-1\}$, $Plex(\{x,y,z\})$, $Grlex(\{x,y,z\})$, $\{z-1, y-1, x-1\}$
3. $\{z+1, y+1, x+1\}$, $Plex(\{x,y,z\})$, $Grlex(\{x,y,z\})$, $\{z+1, y+1, x+1\}$
4. $\{z-1, y-1, x+2\}$, $Plex(\{x,y,z\})$, $Grlex(\{x,y,z\})$, $\{z-1, y-1, x+2\}$
5. $\{z-1, y^2+3*y+1, x+y+1\}$, $Plex(\{x,y,z\})$, $Grlex(\{x,y,z\})$, $\{z-1, x+y+1, y*z-y, y^2+3*y+1, x*y-2*y-1\}$
6. $\{z^6-z^2, z^2+y, x+z\}$, $Plex(\{x,y,z\})$, $Grlex(\{x,y,z\})$, $\{x+z, z^2+y, x*z-y, x*y+y*z, y*z^2+y^2, y^3-y, x*y*z-y^2, x*y^2+y^2*z, y^2*z^2+y, y^3*z-y*z, x*y^2*z-y\}$

Все тесты пройдены

3 Benchmark

Время работы программы было замерено на тестовых полиномах стандартным способом. Ниже указано время работы программы для каждого тестового примера в том же порядке, в котором они расположены выше.

1. 1.215 second
2. 0.162 second
3. 0.162 second
4. 0.162 second
5. 0.658 second
6. 4.572 second

Тестирование проводилось на Intel(R) Core(TM) i3-6100 CPU @ 3.70 GHz с 8 Гб оперативной памяти.

4 Выводы

У данной реализации есть три слабых места. Первое - довольно часто приходится решать СЛАУ. MTL4 для решения системы использует LU разложение, которое выполняется за $O(n^3)$. Возможно, данный процесс можно оптимизоровать.

Вторым слабым местом является то, что решение СЛАУ - не всегда вектор, иногда это линейное пространство. В данном случае необходимо взять какой-нибудь не нулевой вектор. Но данная реализация не поддерживает этого. В этом случае алгоритм не корректен.

Третьим слабым местом является поиск нормальной формы полинома. В данной программе реализован поиск нормальной формы поиском остатка от деления, что не особо быстро. Исправлением данной проблемы служит алгоритм, предложенный в статье, который не использует деление.

Так же имеются проблемы с реализацией, например частое клонирование объектов, что так же может ухудшить скорость работы программы.