МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра информационных технологий**

**КУРСОВАЯ РАБОТА**

**СУПЕРСЖАТИЕ ДАННЫХ. АЛГОРИТМ СИМВОЛЬНОЙ РЕГРЕССИИ**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_М.И. Мичков

(подпись)

Направление подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем   курс   3

(код, наименование)

Направленность (профиль) Технологии программирования

Научный руководитель

д-р физ.-мат. наук, проф. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.И. Миков

(подпись)

Нормоконтролер

ст. преп. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.В. Харченко

(подпись)

Краснодар

2022

**РЕФЕРАТ**

Курсовая работа 34 с., 2 рис., 1 табл., 7 источников, 1 прил.

СИМВОЛЬНАЯ РЕГРЕССИЯ, РЕГРЕССИОННЫЙ АНАЛИЗ, МЕТОД ВЕТВЕЙ И ГРАНИЦ, МЕТОД НАПРАВЛЕННОГО ПОИСКА

Объектом исследования является задача символьной регрессии.

Цель работы – разработка алгоритма символьной регрессии, позволяющий находить простые формулы для зависимостей во входных данных.

В результате работы был разработан алгоритм, позволяющий по набору входных данных находить формулу, ему соответствующую. Возвращаемые данным алгоритмом формулы обладают значительно меньшей сложностью, чем формулы, возвращаемые другими известными алгоритмами, что позволяет во многих случаях находить точные формулы, описывающие входные данные.

Содержание

[Введение 4](#_Toc103943705)

[1 Постановка задачи 5](#_Toc103943706)

[2 Существующие алгоритмы 7](#_Toc103943707)

[3 Общая схема алгоритма 9](#_Toc103943708)

[4 Представление формул в виде деревьев 12](#_Toc103943709)

[4.1 Представление в виде последовательности 15](#_Toc103943710)

[4.2 Вычисление деревьев 16](#_Toc103943711)

[5 Алгоритм Левенберга-Марквардта 19](#_Toc103943712)

[6 Генерация деревьев 22](#_Toc103943713)

[7 Префиксное дерево 24](#_Toc103943714)

[8 Фронт Парето 26](#_Toc103943715)

[8.1 Приоритетная очередь Парето 27](#_Toc103943716)

[9 Критерий остановки алгоритма 29](#_Toc103943717)

[10 Экспериментальное исследование алгоритма 31](#_Toc103943718)

[Заключение 34](#_Toc103943719)

[Список используемых источников 35](#_Toc103943720)

[Приложение А 36](#_Toc103943721)

# Введение

Результатом научных экспериментов часто являются большие наборы данных, представляющие собой наборы значений некоторых взаимосвязанных переменных. Целью научного исследования чаще всего является получение эмпирической зависимости между переменными, построенной на основе экспериментальных данных, что позволяет в дальнейшем предсказывать значения некоторых переменных на основе остальных без проведения эксперимента. Нахождение такой зависимости называется регрессией.

Существует множество широко известных типов регрессии, полученных на основе статистики: линейная регрессия, полиномиальная регрессия, нелинейная регрессия и другие. Недостатком таких типов регрессии является то, что для их применения необходимо знать форму зависимости между переменными заранее: статистические методы возможно использовать лишь для нахождения коэффициентов в формуле зависимости.

Для решения задачи регрессии также часто применяются нейросетевые модели: нейронную сеть некоторой архитектуры обучают на наборе входных данных и используют для предсказания выходных значений. Нейросетевые методы также обладают некоторыми недостатками, в частности для них необходимы большие наборы обучающих примеров, а также результатом обучения является сама нейронная сеть, предсказывающая значения переменных, но которую невозможно представить в человекочитаемом виде, в отличие от моделей статистических методов.

В связи с этим, важной задачей является задача *символьной регрессии*, заключающаяся в нахождении человекочитаемой формулы, соответствующей набору входных данных, но не требующей предварительного определения пользователем формы зависимости между переменными. В данной курсовой работе будет представлен детерминированный алгоритм ее решения, оптимизированный для нахождения самой простой возможной формулы зависимости.

1 Постановка задачи

Задача символьной регрессии заключается в поиске явной формулы, лучше всего описывающий некоторый входной набор данных. Входной набор данных представляется в виде набора из точек -мерного пространства . Целью символьной регрессии является поиск функциональных зависимостей вида

где функции выражены в виде композиции элементарных функций и операций, указанных заранее. Далее такое представление функций будет называться *формулами*.

Данные функциональные зависимости должны выполняться для всех точек из входного набора данных. В частности, – это -я точка из набора данных, с , а – это ее -я координата с . Используя обозначения выше, полную форму функциональных зависимостей можно записать как

Задача символьной регрессии заключается в поиске формул для , для которых уравнение выше выполняется для всех .

Задачи поиска каждой из формул являются идентичными, в связи с чем в дальнейшем будет рассматриваться задача поиска только одной из них. Для этого, введем новые обозначения. Пусть координата, для которой ищется завимимость () обозначается , а все остальные обозначаются с . Тогда задача символьной регрессии одной функции сводится к поиску , удовлетворяющей

Поскольку поиск точной формулы зачастую является невозможным, рассматривается задача поиска зависимости с погрешностью, а именно

где – неизвестная погрешность на -й точке из набора данных. Необходимо, чтобы эта погрешность была как можно меньше, для чего должна быть введена общая величина «ошибки» для всего набора, находимая как сумма квадратов всех погрешностей:

Целью символьной регрессии является нахождение такой формулы , что . В случае, если точная зависимость входных данных действительно представима через элементарные функции, то возможно найти с . В случае же если в исходных данных нет такой зависимости, или же она является примерной, то достичь невозможно, но все еще может быть возможно получить функции с достаточно малыми значениями .

2 Существующие алгоритмы

Для решения поставленной выше задачи возможно применение многих различных алгоритмов. Ниже представлены некоторые из них.

1. *Генетические алгоритмы* [1][2] – это алгоритмы, имитирующие процесс эволюции для своей работы. В них каждая проверяемая формула представляется в виде некой последовательности чисел (генома), и рассматривается как «особь». Одновременно рассматривается большое число особей, называемых вместе «популяцией». В данной популяции проводится естественный отбор на основе значения для каждой из особей, и самые лучшие особи используются для скрещивания и создания «потомства». К потомству также применяются «мутации», случайным образом изменяющие их геном. Данный процесс повторяется большое число раз, что ведет к постепенному улучшению качества формул в популяции. Детали реализации каждого из этих шагов различны для разных генетических алгоритмов. Генетические алгоритмы – самый часто используемый класс алгоритмов для символьной регрессии. Генетические алгоритмы обладают некоторыми недостатками: они достаточно часто предлагают излишне сложные выражения даже для очень простых задач. Такие выражения часто также не являются человекочитаемыми.
2. *Deep Symbolic Regression* [3] – алгоритм, основанный на рекуррентной нейронной сети. В нем нейронная сеть, обученная на различных формулах, предлагает дальнейшие возможные улучшения для них, что позволяет исследовать пространство возможных формул. Данный алгоритм также обладает теми же недостатками, что и генетические алгоритмы, а именно излишней сложностью возвращаемых выражений.
3. *AIFeynman* [4] – алгоритм, основанный на методе «разделяй и властвуй», обучающий нейронную сеть на входных данных для того, чтобы определить их свойства, например, возможность разделения переменных, и рекурсивно рассматривает все возникающие подзадачи. Данный алгоритм специализируется на нахождении зависимостей, получаемых в результате физических экспериментов, в связи с чем не подходит для общего использования.
4. *Bayesian Symbolic Regression* [5] – алгоритм, основанный на получении случайной выборки формул с распределением вероятности, определенным при помощи входных данных и теоремы Байеса. Выборка случайных формул в соответствии с распределением, не заданном в явном виде, происходит при помощи алгоритма Монте-Карло для цепей Маркова. К сожалению, по результатам исследования, данный алгоритм не обладает достаточной точностью, чтобы его было возможно использовать для решения представленной задачи.

Все вышеперечисленные алгоритмы обладают некоторыми недостатками, в частности возможностью нахождения формул, намного более сложных, чем это действительно необходимо для нахождения точной зависимости. В связи с этим в данной курсовой работе будет представлен новый алгоритм, *всегда* выдающий самое простое представление функциональной зависимости, если таковое возможно. Данный алгоритм основан на уже существующем алгоритме, называемом *Prioritized Grammar Enumeration* [6], но является более простым, стабильным и обладает более высокой скоростью работы.

3 Общая схема алгоритма

Задача поиска самой простой формулы, удовлетворяющей набору из условий (по одному на каждую точку входного набора), является NP-сложной. В связи с этим (при условии ) нахождение самой простой такой формулы можно гарантировать лишь при помощи перебора всех возможных формул. Для ускорения такого поиска часто используется *метод ветвей и границ* и подобные, позволяющие осуществлять направленный поиск с использованием некоторой эвристики. Именно к этому классу и относится рассматриваемый алгоритм.

В этом алгоритме формулы представлены в виде *деревьев выражений*, где узлами являются операции, функции и переменные. В каждом дереве также имеется множество произвольных параметров. Каждое такое дерево можно вычислить, указав значения всех переменных и параметров, и получить его численное значение. Вычислив дерево на всех точках из входного набора данных, возможно вычислить , что определит точность, с которой формула соответствует входному набору данных.

Каждое дерево выражения также может быть представлено в виде последовательности целых чисел. Длина этой последовательности может быть представлена как мера сложности формулы, которая представляется деревом.

Для подбора параметров, лучше всего соответствующих входному набору данных, используется алгоритм нелинейной регрессии, а именно *алгоритм Левенберга-Марквардта* [7]. Для его работы необходимо вычисление частных производных вычисляемой функции от всех параметров, для чего используется техника *автоматического дифференцирования*.

Новые деревья выражений генерируются на основе предыдущих деревьев в направлении увеличения сложности. Таким образом поиск никогда не возвращается назад к уже рассмотренным деревьям меньшей сложности. Тем не менее, некоторые деревья можно получить несколькими способами из других, в связи с чем для ускорения работы алгоритма используется *мемоизация* при помощи *префиксного дерева*, хранящего все рассмотренные деревья выражений в формате, позволяющем быстро проверить наличие дерева выражения в префиксном дереве.

Поиск направляется при помощи *приоритетной очереди Парето*, осуществляющей сортировку всех деревьев по двум параметрам одновременно: точности и сложности. На каждом шаге из очереди выбираются все деревья, находящиеся на *фронте Парето*, т.е. деревья, которые лучше всех остальных хотя бы по одному параметру. Также поддерживается фронт Парето всех рассмотренных деревьев для нахождения лучших.

Поскольку для разных применений данного алгоритма баланс между требуемой точностью и допустимой сложностью формул может быть различным, алгоритм является интерактивным: он периодически показывает пользователю все лучшие формулы (т.е. фронт Парето), которые он нашел за время своей работы. Если пользователь считает, что достаточно хорошая формула уже найдена, он может остановить работу алгоритма.

Несмотря на это, возможно использование опционального *критерия остановки алгоритма*, позволяющего определять момент нахождения правильной формулы программным способом. Данный критерий не является абсолютно точным и может останавливаться слишком рано или вообще не останавливаться для некоторых наборов данных, но для большей их части он позволяет найти правильную формулу без участия человека.

Таким образом, алгоритм можно представить следующим образом:

1. сгенерировать все начальные деревья выражений и поместить их в очередь;
2. получить дерево из очереди;
3. сгенерировать новые деревья на основе данного;
4. для каждого из новых деревьев выполнить следующее:
   1. преобразовать дерево в последовательность чисел (сериализация);
   2. проверить, сохранено ли это дерево в префиксном дереве. Если да, то его можно пропустить;
   3. сгенерировать случайные значения параметров и использовать алгоритм Левенберга-Марквардта для нахождения лучших значений параметров. Повторить для нескольких случайных наборов параметров (например, 10) и выбрать лучшие;
   4. если данная формула точно описывает входной набор данных (т.е. меньше некоторого малого ), то остановить работу программы и вернуть данную формулу;
   5. иначе поместить дерево вместе с его характеристиками в очередь, во фронт Парето, а также в префиксное дерево, и перейти к следующему дереву;
5. периодически показывать пользователю фронт Парето из лучших найденных на данный момент формул. Если пользователь считает, что необходимая точность уже достигнута, прервать выполнение алгоритма. Иначе повторять шаги 2-4 для новых деревьев. Вместо этого также возможно применение критерия остановки алгоритма.

В следующих разделах все вышеописанные компоненты будут рассматриваться подробнее.

4 Представление формул в виде деревьев

Каждую формулу можно представить в виде дерева. Поскольку поиск осуществляется методом перебора, необходимо как можно сильнее уменьшить количество различных типов узлов, а также уменьшить количество эквивалентных друг другу формул. Далее будет описан пример представления, удовлетворяющего этим требованиям.

Существует 3 типа узлов: узлы-переменные, унарные операции (функции) и -арные операции.

* Узлы-переменные (VarNode) содержат только номер переменной, которую они представляют. Например, узел, обозначающий хранит число 1.
* Унарные операции (UnaryNode) содержат тип операции, которую они представляют, а также указатель на узел, обозначающий его аргумент. Поддерживается 8 типов унарных операций, Inverse, обозначающий , Exp, обозначающий , а также Ln, Sqrt, Sin, Cos, Asin, Acos, обозначающие соответственно Таким образом, например, формула будет представлена как унарный узел типа Exp, содержащий ссылку на узел, представляющий формулу . Эти 8 типов операций также называются базисными функциями и пользователю предоставляется возможность выбора набора используемых.
* -арные операции (BigNode) содержат тип операции, а также набор указателей своих составляющих. Поддерживаются 2 типа -арных операций, а именно AddConst, представляющий сумму, и MultiplyConst, представляющий произведение. Обе этих операции также содержат в себе параметр, определяющий или свободный член в сумме, или коэффициент перед произведением. Таким образом, формула будет представляться как узел типа MultiplyConst, содержащий ссылки на узлы , и .

Для обеспечения уникальности представления формул, на деревья наложены дополнительные ограничения.

1. Для деревьев определено сравнение, чтобы их всегда можно было распределить по порядку. Сортировка определена следующим образом:
   * если выражения имеют разный вид, то сортировка соответствует VarNode < UnaryNode < BigNode;
   * при сравнении двух VarNode сравниваются номера их переменных, с если ;
   * при сравнении двух UnaryNode считается что Inverse <  
     < Exp < Ln < Sqrt < Sin < Cos < Asin < Acos. Если же типы операций одинаковые, то сравниваются их аргументы;
   * при сравнении двух BigNode считается что AddConst <  
     < MultiplyConst. Если же они одного типа, то сравнивается их количество потомков – узел с меньшим числом потомков считается меньше. Если же потомков одинаковое число, то они распределяются по возрастанию и сравниваются попарно;
2. Допустимым представлением BigNode считается только то, где его потомки отсортированы по возрастанию. Таким образом, например,  
    является допустимым представлением, в то время как  
    недопустимо.
3. Внутри каждого AddConst допустимы только узлы MultiplyConst. Это обеспечивает что перед каждым слагаемым будет стоять коэффициент. Благодаря этому выражение вида недопустимо, допустимо лишь .
4. Корнем дерева выражения может являться только AddConst. Таким образом выражение вида недопустимо, допустимо только  
   . Таким образом самой простой возможной формулой является линейная зависимость от одной из переменных.
5. Потомками MultiplyConst не могут быть другие MultiplyConst и AddConst. Это означает что допустимы только выражения с полностью раскрытыми скобками.
6. В AddConst недопустимы одинаковые потомки. Поскольку перед каждым слагаемым имеется коэффициент, возможно привести подобные. Таким образом, допустимой формой является с . Подобных ограничений на MultiplyConst не накладывается, поскольку .
7. В каждом MultiplyConst допустим только один Inverse, только один Exp и только один Sqrt. Это необходимо, поскольку , и .
8. В каждом Exp внутри AddConst параметр должен быть деактивирован (приравнен к 0), поскольку с .
9. В каждом Inverse внутри первой MultiplyConst параметр должен быть деактивирован (приравнен к 1), поскольку

где и . При условии когда , слагаемое можно убрать.

1. В каждом Inverse недопустимы другие Inverse, если между ними нет других функций (например Exp), поскольку

В то же время выражения вида допустимы.

C учетом вышеописанных ограничений, в BigNode было добавлено поле constDisabled для учета возможности блокировки параметра, а также для хранения потомков BigNode была реализована собственная структура данных SmallSet.

Потомки BigNode должны храниться в отсортированном виде, причем в случае MultiplyNode возможны повторяющиеся потомки. Из-за этого стандартная структура данных std::set языка C++ не подходит для данной задачи. Кроме того, поскольку в каждом BigNode будет храниться около 2-5 потомков, необходима структура данных, работающая быстро при малом количестве данных, а не асимптотически. В связи с этим SmallSet представляет собой простой отсортированный std::vector с проверкой на уникальность элементов для AddNode. Также для него была реализована поддержка итераторов, и возможность удаления и вставки элементов.

## 4.1 Представление в виде последовательности

Также необходима возможность представить любую формулу в виде последовательности целых чисел (так называемая *сериализация*). Для представленного алгорима обратное преобразование (десериализация) не является необходимым, но может быть легко реализовано.

Сериализация выражения происходит рекурсивно в порядке обхода в глубину. Узлы дерева представляются следующим образом:

1. узел AddConst представляется как 1, потомки, 0, т.е. число 1, обозначающее AddConst, далее идет представление всех потомков по порядку, и представление завершается числом 0;
2. узел MultiplyConst представляется как 2, потомки, 0, аналогично AddConst;
3. узел Inverse представляется как 3, потомок, т.е. число 3, после чего идет представление единственного потомка;
4. узел Exp представляется как 4, потомок, аналогично Inverse;
5. узлы Ln, Sqrt, Sin, Cos, Asin, Acos представляются аналогично с кодами 5, 6, 7, 8, 9, 10 соответственно
6. узел VarNode переменной представляется как , т.е. представляется как 11, как 12 и т.д.

Таким образом, в соответствии с этими правилами выражение представляется как 1,2,11,0,0, а более сложное выражение

представляется как 1,2,11,0,2,12,3,1,2,11,11,0,2,12,12,0,0,

0,0. Как мы можем заметить, в конце представления появляется большое число нулей, которые не важны для записи выражения. В связи с этим, все нули с конца строки возможно убрать, тогда представления будут соответственно 1,2,11 и 1,2,11,0,2,12,3,1,2,11,11,0,2,12,12, длин 3 и 15.

## 4.2 Вычисление деревьев

Для вычисления значения дерева необходимы значения всех переменных, , а также значения всех параметров . Нумерация параметров в дереве выполняется в порядке рекурсивного обхода в глубину (preordering), пропуская все деактивроиванные параметры.

Тогда при вычислении дерева происходит рекурсивный обход его в глубину и последовательное вычисление всех подвыражений, в него входящих. При этом поддерживается индекс следующего параметра, а все вычисленные значения промежуточных подвыражений сохраняются.

Для алгоритма Левенберга-Марквардта также требуется поиск частных производных выражения по каждому из его параметров, для поиска которых используется техника *автоматического дифференцирования*. Эта техника основана на правиле производной сложной функции и работает следующим образом: в порядке рекурсивного обхода в глубину для каждого из узлов (подвыражений) дерева вычисляется частная производная   
 – производная всего выражения по «переменной» . Для примера рассмотрим выражение

представленное деревом на рисунке 1. Далее будет приведен пример вычисления частных производных .

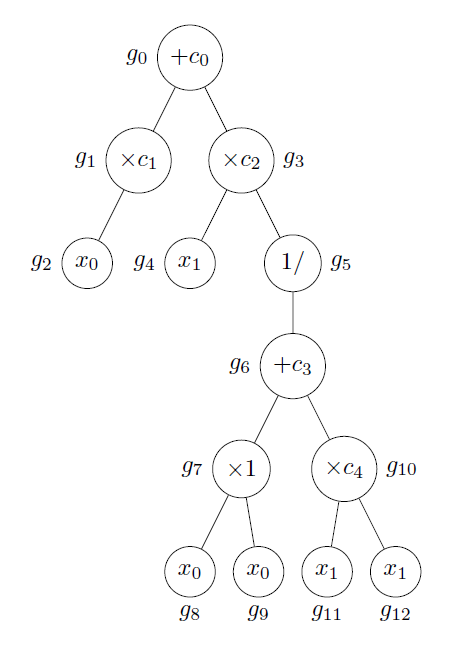


Рисунок 1 – Дерево выражения

Таким образом можно постепенно вычислить все частные производные. Нам потребуется лишь 5 из них, содержащих параметры, а именно

Тогда найти производные от параметров не составляет труда:

5 Алгоритм Левенберга-Марквардта

В проверяемых алгоритмом формулах присутствуют свободные параметры, которые необходимо подобрать для обеспечения наиболее точного соответствия входным данным. Таким образом, необходимо подобрать числа чтобы функция наиболее точно соответствовала набору исходных данных, т.е. была минимизирована функция

или, в векторных обозначениях,

обозначающая сумму квадратов отклонений от входных данных. Это – так называемая задача *нелинейной регрессии*, поскольку зависит от параметров нелинейно.

Существует множество методов, решающих данную задачу, например, метод Ньютона и метод градиентного спуска, но в данной работе был выбран *алгоритм Левенберга-Марквардта*, обладающий высокой скоростью сходимости. Алгоритм можно записать следующим образом:

1. выбираются произвольные начальные значения параметров ;
2. устанавливается начальное значение (в программе выбрано );
3. вычисляются значения функции на всем наборе входных данных;
4. находятся градиенты функции по параметрам на каждой из входных точек . Градиенты воспринимаются как матрица размерности ;
5. вычисляется значение ошибки . Если значение ошибки , то необходимая точность достигнута и алгоритм прерывается;
6. если новое значение ошибки меньше значения ошибки из предыдущей итерации алгоритма (считается что ), то значение заменяется на , иначе ;
7. из системы линейных уравнений ниже находится , где  
    – единичная матрица ;
8. если , то производится его нормирование: ;
9. если выполняется для всех , то требуемая точность достигнута и выполнение алгоритма прекращается;
10. происходит обновление параметров и алгоритм переходит к шагу 3.

Для решения системы линейных уравнений используется метод Гаусса. Также, поскольку алгоритм Левенберга-Марквардта является итеративным и имеет элемент случайности (генерация начальных значений параметров), в основном алгоритме он выполняется 10 раз на различных начальных значениях параметров, после чего из всех конечных значений параметров выбираются наилучшие (обладающие наименьшей ).

6 Генерация деревьев

Для работы алгоритма требуется, чтобы из любого дерева выражения можно было получить множество новых, похожих, деревьев, каждое из которых будет чуть «сложнее» изначального. При этом все новые деревья обязаны соответствовать всем требованиям, описанным в разделе 4. Пример такого алгоритма генерации деревьев будет описан ниже.

К каждому узлу дерева можно применить несколько различных преобразований. Например, имея узел суммы AddConst возможно добавить к нему новое слагаемое:

для всех . Возможна ситуация, что одна из также имеет вид , тогда подобная подстановка невозможна.

Аналогичным образом можно определить преобразования узла произведения MultiplyConst:

Будем считать, что к узлам UnaryNode преобразования применить невозможно. Тогда оставшиеся преобразования – преобразования «листьев» дерева, т.е. узлов VarNode. Поскольку VarNode всегда содержатся внутри MultiplyConst и AddConst, единственные возможные типы узлов, на которые их возможно заменить – это Inverse и Exp. В связи с этим возможны лишь следующие замены:

Данные 10 типов подстановок позволяют сгенерировать любую формулу, представимую в виде дерева, если в качестве начальных взять простейшие формулы вида . Таким образом, для нахождения всех новых деревьев, построенных на основе данного, достаточно к каждому из узлов применить все возможные подстановки, и полученные в результате каждой из подстановок деревья будут являться результатом генерации. Например, из формулы подстановками могут быть получены следующие формулы (для упрощения используются только Inverse и Exp):

Также, из-за ограничения на единственный узел Inverse и единственный узел Exp в каждом MultiplyConst, соответствующие подстановки не применяются если в MultiplyConst уже имеются Inverse или Exp.

7 Префиксное дерево

Несмотря на направленность поиска, одинаковые формулы могут быть получены несколько раз при помощи разных подстановок, например формула может быть получена и из , и из . В связи с этим, необходимо запоминать, какие деревья уже были рассмотрены, а какие нет, чтобы не повторять вычисления для одной и той же формулы несколько раз. Для запоминания деревьев необходима структура данных, которая позволит быстро определять, присутствует ли данное дерево в структуре, или нет, а также эффективно использующая память. Такой структурой является *префиксное дерево*.

Префиксное дерево используется для сохранения произвольного множества строк в некотором алфавите . Каждый узел дерева соответствует единственной возможной строке, определяемой полностью его положением в дереве. Каждый узел также содержит указателей на своих потомков (возможно отсутствующих), каждый из которых соответствует одному символу алфавита. Таким образом, корень дерева соответствует пустой строке, все его потомки – односимвольным строкам, символ в которых определяется их положением в списке потомков корня дерева, а все их потомки – двухсимвольными строками и т.д. При этом каждый узел в дереве также содержит boolean поле, содержащее true если данная строка присутствует в хранимом множестве и false если не присутствует.

Поскольку каждое дерево выражения можно представить последовательностью чисел от 0 до ( – число переменных), можно сказать, что эти последовательности являются строками в алфавите размера . Таким образом, множество деревьев выражений можно хранить в префиксном дереве. В качестве примера, на рисунке 2 представлено префиксное дерево, хранящее формулы , , ,  
, (представленные последовательностями 1,2,5, 1,2,6, 1,2,5,6, 1,2,6,5, 1,2,4,5).

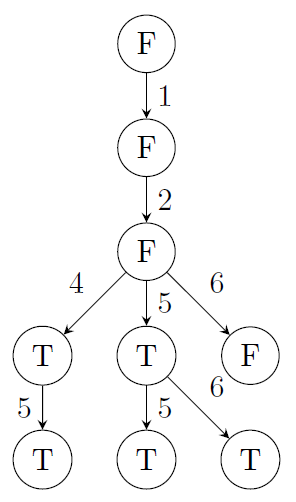


Рисунок 2 – Префиксное дерево (T обозначает true, F – false)

Проверка наличия строки в префиксном дереве определяется просто проходом по соответствующим указателям (если они имеются), и проверке значения boolean поля в узле. Добавление новой строки осуществляется проходом по существующим указателям, а затем созданием недостающих узлов. Обе эти операции выполняются за время, линейно зависящее от длины строки, т.е. от сложности формулы.

8 Фронт Парето

Искомая формула должна быть оптимизирована одновременно по 2 параметрам: точности, характеризуемой , и сложности, характеризуемой длиной последовательности чисел, соответствующей данной формуле (далее длина будет обозначаться ). В связи с этим, часто будет возникать ситуация, когда одна формула лучше другой по точности, но хуже по сложности:  
. Поскольку точно выразить требумый баланс между и не представляется возможным, а также поскольку требуемый баланс будет отличаться для каждой рассматривамой задачи, предлагается использовать *оптимальность по Парето* – формула считается оптимальной, если не существует других формул, которые лучше ее по обоим параметрам. Когда формула лучше другой по обоим параметрам, говорят, что она *доминирует ее по Парето*. Таким образом будет получен целый *фронт Парето* – множество оптимальных по Парето решений, которые не доминируются никакими другими решениями.

Поскольку является целочисленным параметром (и при этом достаточно малым по величине), можно рассматривать фронт Парето более простым способом: для каждого значения параметра (для каждой «весовой категории») можно найти решение с наименьшей . При этом, если при увеличении не достигается уменьшение , то данное решение не является оптимальным по Парето и не добавляется во фронт.

Таким образом фронт Парето можно представить как std::vector из формул вместе с их значениями . Индексом формулы является ее сложность . При добавлении новой формулы во фронт Парето необходимо лишь сравнить ее с формулы, уже находящейся во фронте с той же самой сложностью. Если новая формула имеет меньшую , чем старая, то старая заменяется на новую. При необходимости, возможно получить все формулы из фронта Парето, исключив те из них, которые не оптимальны по Парето, т.е. которые имеют большее чем предыдщуие формулы во фронте, но не имеют при этом меньшую .

## 8.1 Приоритетная очередь Парето

Частым компонентом алгоритмов направленного поиска является *приоритетная очередь* – очередь, где элементы отсортированы по какому-либо параметру, и из очереди всегда первыми достаются элементы с самым большим значением параметра. Тем не менее, подобную очередь невозможно применить, если необходимо учитывать два и более параметров, в связи с чем в данном алгоритме будет использована приоритетная очередь, основанная на оптимальности по Парето, или *приоритетная очередь Парето* (Pareto Priority Queue).

В данной очереди ее элементы располагаются по последовательным фронтам Парето. Первый (основной) фронт Парето содержит те элементы, которые не доминирются никакими другими элементами. Второй фронт Парето содержит элементы, которые доминируются только элементами из первого фронта, третий фронт содержит элементы, которые доминируются только элементами из первого и второго фронта и т.д. Основными операциями приоритетной очереди является добавление нового элемента в нее, а также извлечение первого фронта Парето целиком.

Поскольку в рассматриваемой задаче один из параметров () является целочисленным, то структуру приоритетной очереди Парето можно значительно упростить. В частности, приоритетная очередь Парето будет содержать по одной приоритетной очереди (по параметру ) на каждое значение параметра . При этом лучшие по элементы будут находиться на вершине каждой из очередей, и будут вместе образовывать фронт Парето (с дополнительным ограничением на случай , как и ранее). Таким образом извлечение фронта Парето из приоритетной очереди заключается просто в последовательном извлечении элементов из очередей, при условии что их лучше предыдущего элемента фронта.

9 Критерий остановки алгоритма

Для полной автоматизации решения задачи символьной регрессии, необходимо заменить шаг проверки человеком фронта Парето для определения возможности остановки алгоритма на автоматическую проверку. Для этого необходимо определить, какими свойствами, заметными в выходных фронтах Парето, обладают правильные формулы, но не обладают неправильные. По результатам наблюдений, это свойство – это резкое уменьшении ошибки при малом увеличении сложности. Пока правильная формула еще не была найдена, ошибка формул из фронта Парето уменьшается медленно и равномерно: при увеличении сложности на 1 ошибка уменьшается примерно в раза. В то же время, когда алгоритм находит правильную формулу, ее ошибка уменьшается в раз по сравнению с формулами сложности на 1 ниже.

Для определения момента нахождения правильной формулы, было введено понятие *наклона фронта Парето*, определяемого следующим образом:

В ней используются логарифмы ошибки, поскольку при увеличении сложности ошибка уменьшается экспоненциально. Логарифмы позволяют свести зависимость к примерно линейной.

В рассмотренных ранее ситуациях, падение ошибки в раза определяет наклон равный , в то время как падение ошибки в раз определяет наклон в . В качестве порогового значения было подобрано значение наклона в 1.5, соответствующее падению ошибки в 4.5 раза. При условии, что наклон для какой-то из формул во фронте Парето оказывается больше 1.5, то поиск останавливается, и эта формула возвращается в качестве результата работы алгоритма. Если же ни для одной из формул наклон не больше 1.5, работа алгоритма продолжается.

10 Экспериментальное исследование алгоритма

Точность решения задачи символьной регрессии оценивать сложно, поскольку алгоритм либо находит правильную закономерность во входных данных, либо не находит. В случаях, когда во входных данных не было ярко выраженной закономерности, или если во входных данных присутствуют шумы, то благодаря фронту Парето всегда можно пожертвовать простотой формулы ради дополнительного увеличения точности.

В связи с этим, основной анализируемой характеристикой в данной работе является скорость работы алгоритма, выражаемая во времени его работы, а также количестве рассмотренных функций (и количестве их вычислений). Все тесты проводились со входным набором, состоящим из точек. В некоторых тестах во входные данные добавлена погрешность, являющаяся нормально распределенной случайной величиной с некоторой . Базисными функциями были выбраны только Inverse и Exp. Результаты тестов приведены в таблице 1. В качестве входных данных алгоритму также были представлены некоторые функции, которые невозможно точно представить в виде формулы с использованием лишь арифметических операций и экспоненты. Несмотря на это, алгоритм для многих из них нашел достаточно близкую приближенную формулу.

Таблица 1 – Результаты исследования

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Формула |  |  | Время (c) | Количество рассмотренных формул | Ошибка |
|  | 0 |  | 98 | 2611 | 0 |
| 0.001 | 130 | 3476 | 0.001 |
|  | 0 |  | 0 | 13 | 0 |
| 0.01 | 4 | 223 | 0.01 |
| 0.1 | 4 | 232 | 0.1 |

Продолжение таблицы 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 |  | 0 | 8 | 0 |
| 0.001 | 0 | 8 | 0.001 |
| 0.01 | 5 | 257 | 0.01 |
| 0.1 | 5 | 260 | 0.1 |
|  | 0 |  | 1 | 60 | 0 |
| найдено | 0 |  | 1 | 60 | 4.5e-06 |
| найдено | 0 |  | 0 | 32 | 8.6e-08 |
| найдено |  | 0 | 43 | 2.8e-07 |
| найдено | 0 |  | 1 | 81 | 8e-07 |
| найдено |  | 108 | 2483 | 0.0227 |
| найдено | 0 |  | 0 | 11 | 4.1e-07 |
| найдено |  | 1 | 67 | 8.2e-07 |
|  | не удалось найти формулу | | | | |

Продолжение таблицы 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 |  | 49 | 2093 | 0 |
|  | 0 |  | 12 | 616 | 0.00043 |

Для каждой из рассмотренных функций, алгоритм Левенберга-Марквардта применяется 10 раз, и каждый его запуск вычисляет значения функции на всех точках около 50-100 раз. Таким образом, каждая из рассмотренных алгоритмом функций была им вычислена примерно 50000-100000 раз. Также можно заключить, что алгоритм в среднем рассматривает примерно 25 функций в секунду (25000000 вычислений в секунду), и при этом работает одинаково эффективно на функциях 1, 2 и 4 переменных.

# Заключение

В результате выполнения курсовой работы был получен алгоритм решения задачи символьной регрессии, позволяющий производить поиск наиболее простых форм записи зависимостей, представленных во входных данных. Поиск может осуществляться в интерактивном или автоматизированном режиме.

Было также проведено исследование полученного алгоритма, заключающееся в измерении времени нахождения правильной формулы зависимости, а также ошибки соответствия формулы данным. По результатам исследования, полученный алгоритм обладает приемлемой скоростью работы, способен находить большинство простых зависимостей в данных, в том числе в присутствии помех, а также способен находить хорошие приближения к неизвестным зависимостям с использованием лишь доступных алгоритму элементарных операций.

# Список используемых источников

1 Augusto, D.A. Symbolic regression via genetic programming / D.A. Augusto, H.J.C. Barbosa // Proceedings. Vol. 1. Sixth Brazilian Symposium on Neural Networks. – 2000. – P. 173-178. – DOI 10.1109/SBRN.2000.889734.

2 Schmidt, M. Distilling Free-Form Natural Laws from Experimental Data / M. Schmidt, H. Lipson // Science. – Vol. 324, № 5923. – 2009. – P. 81-85. – DOI 10.1126/science.1165893.

3 Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients / B.K. Petersen, M.L. Larma, T.N. Mundhenk [и др] // arxiv.org : [сайт]. – 2019. – URL: https://arxiv.org/abs/1912.04871 (дата обращения: 03.05.2022).

4 Udrescu, SM. AI Feynman: A physics-inspired method for symbolic regression / SM. Udrescu, M. Tegmark // Science Advances. – Vol. 6, № 16. – 2020. – P. – DOI 10.1126/sciadv.aay2631.

5 Bayesian Symbolic Regression / Y. Jin, W. Fu, J. Kang [и др] // arxiv.org : [сайт]. – 2020. – URL: https://arxiv.org/abs/1910.08892 (дата обращения: 03.05.2022).

6 Worm, T. Prioritized grammar enumeration: symbolic regression by dynamic programming / T. Worm, K. Chiu // GECCO '13: Proceedings of the 15th annual conference on Genetic and evolutionary computation. – 2013. – P. 1021–1028. – DOI 10.1145/2463372.2463486.

7 Гилл, Ф. Практическая оптимизация / Ф. Гилл, У. Мюррей, М. Райт. – М. : Мир, 1985. – 509 с.

# Приложение А

Фрагменты кода программы

// Expression.h

#pragma once

#include <vector>

#include <memory>

#include <random>

#include <map>

#include "SmallSet.h"

using namespace std;

using VariableData = vector<double>;

using ParameterData = vector<double>;

using SerializedExpr = vector<int>;

class ExprNode {

friend class VarNode;

friend class UnaryNode;

friend class BigNode;

protected:

mutable double lastVal;

virtual double eval(const ParameterData& params, const VariableData& data, int& paramIndex) const = 0;

virtual void evalDerivs(const ParameterData& params, ParameterData& derivs, int& paramIndex, double deriv=1) const = 0;

public:

ExprNode\* parent;

inline ExprNode() : parent(nullptr), lastVal(0) {}

inline void setParent(ExprNode\* parent) {

this->parent = parent;

}

virtual unique\_ptr<ExprNode> copy() const = 0;

virtual pair<unique\_ptr<ExprNode>, ExprNode\*> copyAndFind(const ExprNode\* ptr) const = 0;

virtual string toString(const ParameterData& params, int& paramIndex) const = 0;

virtual string toString(int& paramIndex) const = 0;

inline string toString() const {

int paramIndex = 0;

return toString(paramIndex);

}

virtual int countConsts() const = 0;

inline double evalFull(const ParameterData& params, const VariableData& data, ParameterData& derivs) const {

int paramIndex = 0;

double result = eval(params, data, paramIndex);

paramIndex = 0;

evalDerivs(params, derivs, paramIndex);

countEval++;

countEvalDeriv++;

return result;

}

class Iterator : public iterator<forward\_iterator\_tag, ExprNode> {

private:

ExprNode\* ptr;

public:

inline Iterator(ExprNode\* ptr) : ptr(ptr) {}

inline reference operator\*() const { return \*ptr; }

inline pointer operator->() const { return ptr; }

Iterator& operator++();

inline Iterator operator++(int) {

Iterator i = \*this;

++(\*this);

return i;

}

friend inline bool operator==(const Iterator& a, const Iterator& b) { return a.ptr == b.ptr; }

friend inline bool operator!=(const Iterator& a, const Iterator& b) { return a.ptr != b.ptr; }

};

inline Iterator begin() { return Iterator(this); }

inline Iterator end() { return Iterator(nullptr); }

virtual bool operator==(const ExprNode& other) const = 0;

inline bool operator!=(const ExprNode& other) const { return !(\*this == other); };

virtual bool operator<(const ExprNode& other) const = 0;

virtual void serialize(SerializedExpr& result) const = 0;

inline SerializedExpr serialize() const {

SerializedExpr result;

serialize(result);

auto lastNonZero = find\_if(result.rbegin(), result.rend(), [](int x) {return x != 0; }).base();

result.erase(lastNonZero, result.end());

return result;

}

};

static inline bool cmp(const unique\_ptr<ExprNode>& a, const unique\_ptr<ExprNode>& b) { return \*a < \*b; }

class VarNode : public ExprNode {

protected:

virtual double eval(const ParameterData& params, const VariableData& data, int& paramIndex) const;

virtual void evalDerivs(const ParameterData& params, ParameterData& derivs, int& paramIndex, double deriv=1) const;

public:

int varNum;

inline VarNode() : varNum(0), ExprNode() {}

inline VarNode(int varNum) : varNum(varNum), ExprNode() {}

unique\_ptr<ExprNode> copy() const;

virtual pair<unique\_ptr<ExprNode>, ExprNode\*> copyAndFind(const ExprNode\* ptr) const;

string toString(const ParameterData& params, int& paramIndex) const;

string toString(int& paramIndex) const;

virtual int countConsts() const { return 0; }

virtual bool operator==(const ExprNode& other) const;

virtual bool operator<(const ExprNode& other) const;

virtual void serialize(SerializedExpr& result) const;

};

class UnaryNode : public ExprNode {

protected:

virtual double eval(const ParameterData& params, const VariableData& data, int& paramIndex) const;

virtual void evalDerivs(const ParameterData& params, ParameterData& derivs, int& paramIndex, double deriv=1) const;

public:

enum class Type {

Inverse,

Exp,

Sin,

Cos,

Sqrt,

Ln,

Asin,

Acos

};

const static int TOTAL\_TYPES = 8;

Type type;

unique\_ptr<ExprNode> child;

inline UnaryNode(Type type) : type(type), child(), ExprNode() {}

inline UnaryNode(Type type, unique\_ptr<ExprNode> child) : type(type), child(move(child)), ExprNode() {

child->setParent(this);

}

unique\_ptr<ExprNode> copy() const;

virtual pair<unique\_ptr<ExprNode>, ExprNode\*> copyAndFind(const ExprNode\* ptr) const;

string toString(const ParameterData& params, int& paramIndex) const;

string toString(int& paramIndex) const;

virtual int countConsts() const { return child->countConsts(); }

virtual bool operator==(const ExprNode& other) const;

virtual bool operator<(const ExprNode& other) const;

virtual void serialize(SerializedExpr& result) const;

};

class BigNode : public ExprNode {

protected:

virtual double eval(const ParameterData& params, const VariableData& data, int& paramIndex) const;

virtual void evalDerivs(const ParameterData& params, ParameterData& derivs, int& paramIndex, double deriv=1) const;

public:

enum class Type {

AddConst,

MultiplyConst

};

const static int TOTAL\_TYPES = 2;

Type type;

SmallSet<unique\_ptr<ExprNode>, decltype(&cmp)> children;

bool constDisabled;

inline BigNode(Type type) : constDisabled(false), type(type), children(type == Type::AddConst, &cmp), ExprNode() {}

unique\_ptr<ExprNode> copy() const;

virtual pair<unique\_ptr<ExprNode>, ExprNode\*> copyAndFind(const ExprNode\* ptr) const;

string toString(const ParameterData& params, int& paramIndex) const;

string toString(int& paramIndex) const;

virtual int countConsts() const;

virtual bool operator==(const ExprNode& other) const;

virtual bool operator<(const ExprNode& other) const;

virtual void serialize(SerializedExpr& result) const;

};

const static int TOTAL\_TYPES\_IN\_ALL = 1 + UnaryNode::TOTAL\_TYPES + BigNode::TOTAL\_TYPES;

// TreeGenerator.cpp

#include "TreeGenerator.h"

struct AllowData {

bool inverse;

bool exp;

bool sqrt;

};

unique\_ptr<UnaryNode> createUnaryFull(const BigNode& add, UnaryNode::Type type) {

unique\_ptr<ExprNode> addCopy = add.copy();

BigNode& addNode = dynamic\_cast<BigNode&>(\*addCopy);

BigNode& mulNode = dynamic\_cast<BigNode&>(\*addNode.children.front());

mulNode.constDisabled = true;

unique\_ptr<UnaryNode> result = make\_unique<UnaryNode>(type);

addCopy->setParent(result.get());

result->child = move(addCopy);

return result;

}

vector<unique\_ptr<ExprNode>> applyProductionRules(const ExprNode& expr, int varCount, const AllowData& allowed, const BasisFuncs& basis) {

vector<unique\_ptr<ExprNode>> result;

if (dynamic\_cast<const UnaryNode\*>(&expr) != nullptr)

return result;

else if (dynamic\_cast<const BigNode\*>(&expr) != nullptr) {

const BigNode& n = dynamic\_cast<const BigNode&>(expr);

for (int v = 0; v < varCount; v++) {

unique\_ptr<VarNode> var = make\_unique<VarNode>(v);

if (n.type == BigNode::Type::AddConst) {

unique\_ptr<BigNode> big = make\_unique<BigNode>(BigNode::Type::MultiplyConst);

var->setParent(big.get());

big->children.insert(move(var));

unique\_ptr<ExprNode> expr = move(big);

if (n.children.contains(expr)) continue;

unique\_ptr<ExprNode> nCopy = n.copy();

BigNode& nBigCopy = dynamic\_cast<BigNode&>(\*nCopy);

expr->setParent(nCopy.get());

nBigCopy.children.insert(move(expr));

result.emplace\_back(move(nCopy));

}

else {

unique\_ptr<ExprNode> nCopy = n.copy();

BigNode& nBigCopy = dynamic\_cast<BigNode&>(\*nCopy);

var->setParent(nCopy.get());

nBigCopy.children.insert(move(var));

result.emplace\_back(move(nCopy));

}

}

/\*if (n.children.size() > 1) {

for (int i = 0; i < n.children.size(); i++) {

unique\_ptr<ExprNode> nCopy = n.copy();

BigNode& nBigCopy = dynamic\_cast<BigNode&>(\*nCopy);

nBigCopy.children.erase(i);

result.emplace\_back(move(nCopy));

}

}\*/

return result;

}

else if (dynamic\_cast<const VarNode\*>(&expr) != nullptr) {

for (int v = 0; v < varCount; v++) {

unique\_ptr<VarNode> var = make\_unique<VarNode>(v);

unique\_ptr<BigNode> mul = make\_unique<BigNode>(BigNode::Type::MultiplyConst);

unique\_ptr<BigNode> add = make\_unique<BigNode>(BigNode::Type::AddConst);

var->setParent(mul.get());

mul->children.insert(move(var));

mul->setParent(add.get());

add->children.insert(move(mul));

if (basis.inverse && allowed.inverse)

result.emplace\_back(createUnaryFull(\*add, UnaryNode::Type::Inverse));

if (basis.sqrt && allowed.sqrt)

result.emplace\_back(createUnaryFull(\*add, UnaryNode::Type::Sqrt));

if (basis.ln)

result.emplace\_back(createUnaryFull(\*add, UnaryNode::Type::Ln));

if (basis.sin)

result.emplace\_back(createUnaryFull(\*add, UnaryNode::Type::Sin));

if (basis.cos)

result.emplace\_back(createUnaryFull(\*add, UnaryNode::Type::Cos));

if (basis.asin)

result.emplace\_back(createUnaryFull(\*add, UnaryNode::Type::Asin));

if (basis.acos)

result.emplace\_back(createUnaryFull(\*add, UnaryNode::Type::Acos));

if (basis.exp && allowed.exp) {

unique\_ptr<UnaryNode> expNode = make\_unique<UnaryNode>(UnaryNode::Type::Exp);

add->constDisabled = true;

add->setParent(expNode.get());

expNode->child = move(add);

result.emplace\_back(move(expNode));

}

}

return result;

}

}

static void generateTrees(vector<unique\_ptr<ExprNode>>& result, const ExprNode& expr, int varCount, const ExprNode\* n, AllowData allowed, const BasisFuncs& basis) {

if (dynamic\_cast<const UnaryNode\*>(n) != nullptr) {

const UnaryNode\* node = dynamic\_cast<const UnaryNode\*>(n);

allowed.inverse = node->type == UnaryNode::Type::Exp;

allowed.exp = true;

allowed.sqrt = true;

generateTrees(result, expr, varCount, node->child.get(), allowed, basis);

}

else if (dynamic\_cast<const BigNode\*>(n) != nullptr) {

const BigNode\* node = dynamic\_cast<const BigNode\*>(n);

AllowData newAllowed = allowed;

for (const auto& child : node->children) {

if (dynamic\_cast<const UnaryNode\*>(child.get()) != nullptr) {

switch (dynamic\_cast<const UnaryNode\*>(child.get())->type)

{

case UnaryNode::Type::Inverse:

newAllowed.inverse = false;

break;

case UnaryNode::Type::Exp:

newAllowed.exp = false;

break;

case UnaryNode::Type::Sqrt:

newAllowed.sqrt = false;

break;

default:

break;

}

if (dynamic\_cast<const UnaryNode\*>(child.get())->type == UnaryNode::Type::Inverse)

newAllowed.inverse = false;

if (dynamic\_cast<const UnaryNode\*>(child.get())->type == UnaryNode::Type::Exp)

newAllowed.exp = false;

if (dynamic\_cast<const UnaryNode\*>(child.get())->type == UnaryNode::Type::Sqrt)

newAllowed.sqrt = false;

}

}

for (const auto& child : node->children)

generateTrees(result, expr, varCount, child.get(), newAllowed, basis);

}

vector<unique\_ptr<ExprNode>> subTrees = applyProductionRules(\*n, varCount, allowed, basis);

if (n->parent == nullptr) {

result.insert(result.end(), make\_move\_iterator(subTrees.begin()), make\_move\_iterator(subTrees.end()));

return;

}

for (auto& subTree : subTrees) {

auto p = expr.copyAndFind(n);

subTree->setParent(p.second->parent);

if (dynamic\_cast<UnaryNode\*>(n->parent) != nullptr) {

dynamic\_cast<UnaryNode\*>(p.second->parent)->child = move(subTree);

}

else if (dynamic\_cast<BigNode\*>(n->parent) != nullptr) {

BigNode\* bigNode = dynamic\_cast<BigNode\*>(p.second->parent);

for (auto i = bigNode->children.begin(); i != bigNode->children.end(); i++) {

if (i->get() == p.second) {

bigNode->children.erase(i);

bigNode->children.insert(move(subTree));

break;

}

}

}

result.emplace\_back(move(p.first));

}

}

vector<unique\_ptr<ExprNode>> generateTrees(const ExprNode& expr, int varCount, const BasisFuncs& basis) {

vector<unique\_ptr<ExprNode>> result;

AllowData allowed{true, true, true};

generateTrees(result, expr, varCount, &expr, allowed, basis);

return result;

}

// PPQ.h

#pragma once

#include "Expression.h"

#include <queue>

struct PPQNode {

unique\_ptr<ExprNode> tree;

double error;

int complexity;

};

static inline bool cmpError(const PPQNode& a, const PPQNode& b) {

return a.error > b.error;

}

using PPQStack = priority\_queue<PPQNode, vector<PPQNode>, decltype(&cmpError)>;

class PPQ {

private:

vector<PPQStack> stacks;

queue<PPQNode> queue;

void updateQueue();

public:

void insert(unique\_ptr<ExprNode> tree, double error, int complexity);

inline PPQNode pop() {

updateQueue();

PPQNode result = move(queue.front());

queue.pop();

return result;

}

};

struct ParetoData {

string text;

double error;

int complexity;

inline ParetoData() : text(""), error(INFINITY), complexity(0) {}

};

class ParetoFront {

private:

vector<ParetoData> front;

public:

void insert(string text, double error, int complexity);

vector<ParetoData> get() const;

};

// PPQ.cpp

#include "PPQ.h"

#include <iterator>

void PPQ::insert(unique\_ptr<ExprNode> tree, double error, int complexity) {

PPQNode node;

node.tree = move(tree);

node.error = error;

node.complexity = complexity;

int stackIndex = complexity - 3;

while (stacks.size() < stackIndex + 1)

stacks.emplace\_back(&cmpError);

stacks[stackIndex].push(move(node));

}

void PPQ::updateQueue() {

if (!queue.empty()) return;

double prevError = INFINITY;

for (int i = 0; i < stacks.size(); i++) {

if (stacks[i].empty()) continue;

if (stacks[i].top().error < prevError) {

PPQNode moved = move(const\_cast<PPQNode&>(stacks[i].top()));

stacks[i].pop();

prevError = moved.error;

queue.push(move(moved));

//if (queue.size() == 5) break;

}

}

}

void ParetoFront::insert(string text, double error, int complexity) {

int stackIndex = complexity - 3;

if (front.size() < stackIndex + 1)

front.resize(stackIndex + 1);

if (front[stackIndex].text != "" && front[stackIndex].error < error) return;

front[stackIndex].text = text;

front[stackIndex].error = error;

front[stackIndex].complexity = complexity;

}

vector<ParetoData> ParetoFront::get() const {

vector<ParetoData> result;

double prevError = INFINITY;

for (int i = 0; i < front.size(); i++) {

if (front[i].text == "") continue;

if (front[i].error < prevError) {

result.push\_back(front[i]);

prevError = front[i].error;

}

}

return result;

}

// Trie.h

#pragma once

#include <vector>

#include <memory>

using namespace std;

template<typename T>

struct IntegerTrieNode {

vector<unique\_ptr<IntegerTrieNode>> children;

T value;

bool hasValue;

IntegerTrieNode(int n) : children(n), hasValue(false), value() {}

int count() const;

};

template<typename T>

class IntegerTrie {

private:

int radix;

IntegerTrieNode<T> trie;

const IntegerTrieNode<T>\* find(const vector<int>& data) const;

inline IntegerTrieNode<T>\* find(const vector<int>& data) {

return const\_cast<IntegerTrieNode<T>\*>(const\_cast<const IntegerTrie\*>(this)->find(data));

}

public:

IntegerTrie(int radix) : radix(radix), trie(radix) {}

void insert(const vector<int>& data, const T& val);

bool contains(const vector<int>& data) const;

const T& at(const vector<int>& data) const { return find(data)->value; };

T& at(const vector<int>& data) { return find(data)->value; };

inline int count() const { return trie.count(); };

};

// Trie.cpp

#include "Trie.h"

template<typename T>

void IntegerTrie<T>::insert(const vector<int>& data, const T& val) {

IntegerTrieNode<T>\* n = &trie;

for (int x : data) {

if (n->children[x].get() == nullptr)

n->children[x] = make\_unique<IntegerTrieNode<T>>(radix);

n = n->children[x].get();

}

n->hasValue = true;

n->value = val;

}

template<typename T>

const IntegerTrieNode<T>\* IntegerTrie<T>::find(const vector<int>& data) const {

const IntegerTrieNode<T>\* n = &trie;

for (int x : data) {

if (n->children[x].get() == nullptr)

return nullptr;

n = n->children[x].get();

}

return n;

}

template<typename T>

bool IntegerTrie<T>::contains(const vector<int>& data) const {

const IntegerTrieNode<T>\* n = find(data);

if (n == nullptr) return false;

return n->hasValue;

}

template<typename T>

int IntegerTrieNode<T>::count() const {

int result = hasValue ? 1 : 0;

for (const auto& child : children)

if (child)

result += child->count();

return result;

}

template class IntegerTrie<double>;