Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Мухатов Тимур Мухатович

Расширение семантики вероятностных языков программирования оптимизационными запросами

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Рецензент:
д. т. н., профессор Потапов А. С.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Timur Mukhatov

Extension of probabilistic programming language semantics by optimising queries

Bachelor's Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
professor Andrey Terekhov

Reviewer:
professor Alexey Potapov

Saint-Petersburg
2015

Оглавление

Введение

Два ключевых подхода в Общем Искусственном Интеллекте - когнитивная архитектура и универсальный алгоритмический интеллект. Эти подходы начинаются с совершенно разных точек и иногда даже рассматриваются как несовместимые. Однако целесообразно их объединить, чтобы создать искусственный интеллект, который является одинаково и эффективным и общим. Для этого требуется платформа, которая поможет глубоко комбинировать их на концептуальном уровне и уровне реализации.

Вероятностное программирование может стать подходящим основанием для разработки такой платформы. Действительно, с одной стороны, процедуры запросов в Тьюринг-полных вероятностных языках программирования могут использоваться в качестве прямых приближений универсальной индукции и прогноза, которые являются центральными компонентами универсальных интеллектуальных моделей. С другой стороны, вероятностное программирование уже успешно использовалось в познавательном моделировании [10].

Вероятностное программирование - новый мощный инструмент в машинном обучении. Вероятностные языки программирования обеспечивают возможность определить порождающие модели декларативно, без необходимости вручную реализовывать специфичные для определенных моделей алгоритмы вывода, и заменяют их встроенными универсальными методами вывода. Действительно, вероятностные языки очень удобны для использования – достаточно только определить порождающую модель в форме вероятностной программы и наложить желательные условия по которым должны быть вычислены апостериорные вероятности, и эти вероятности будут выведены автоматически.

Много решений в вероятностном программировании используют эффективные методы вывода для определенных типов порождающих моделей (например, Байесовские сети) [3, 11]. Однако полные по Тьюрингу

языки намного более перспективные в контексте общего искусственного интеллекта. Эти вероятностные языки допускают определение порождающих моделей в форме произвольных программ включая программы, которые генерируют другие программы на некотором проблемно-ориентированном языке. Следовательно, вывод по таким порождающим моделям, позволяет определять программы на пользовательском языке. Таким образом один и тот же механизм логического вывода может использоваться для решения очень широкого спектра проблем.

С одной стороны, производительность универсальных методов вывода в вероятностных языках может быть довольно низкой даже для моделей с небольшим количеством случайных выборов [4]. Эти методы обычно основаны на случайном сэмплировании (например, Цепи Маркова Монте-Карло) [5]. Есть некоторые работы в разработке более сильных методов вывода в Тьюринг-полных вероятностных языках, но они не эффективны для всех случаев, например, для вызова программ, хотя в этом направлении достигнут определенный прогресс. Таким образом, необходимы более надлежащие методы вывода, и можно рассмотреть генетическое программирование как подходящего кандидата, так как оно уже было применено к универсальной индукции [8] и познавательной архитектуре [9].

С другой стороны, широкая и простая применимость вывода в вероятностных языках также желательна в эволюционных вычислениях. Действительно, можно было бы применить некоторую существующую реализацию генетического алгоритма просто определив конкретную проблему (которая может быть произвольной проблемой в комбинаторике или индуктивном программировании, и т.д.), не разрабатывая двоичные представления решений или реализовывая специфичную рекомбинацую и операторы мутации.

Следовательно, интересно объединить универсальность вывода по декларативным моделям в Тьюринг-полных вероятностных языках программирования и силу генетического программирования. Эта комбинация

даст универсальный инструмент для быстрого анализа прототипа генетических методов программирования для произвольных предметно-ориентированных языков просто определяя функцию генерации программы на целевом языке. Также она может расширить инструментарий языков, так как стандартный вывод в вероятностном программировании выполняется для удовлетворения условий, в то время как генетическое программирование предназначено для оптимизации функции приспособленности (fitness-функции).

В этой работе я представляю новый подход к алгоритму умозаключения в вероятностных языках программирования на основе генетического программирования, который применяется к трассам выполнения программ. Каждая трассировка программы - экземпляр порождающей модели, определенной программой (результат расширения программы во время оценки). Рекомбинации и мутации трассировок программы гарантируют, что их результаты могут быть сгенерированы первоначальной вероятностной программой. Таким образом трассировки программы используются в качестве "универсального генетического кода" для произвольных порождающих моделей, и достаточно только определить модель в форме вероятностной программы, чтобы выполнить эволюционные вычисления.

Постановка задачи

Целью данной работы является расширение семантики языка Church оптимизационными запросами, используя генетические алгоритмы, а так же сравнение точности работы реализованных запросов на разных задачах.

Для достижения этих целей были поставлены следующие задачи:

- провести обзор существующих алгоритмов умозаключения в вероятностных языках программирования
- изучить принцип работы генетических алгоритмов
- реализовать операторы мутации и скрещивания
- произвести оценку точности запросов на задаче аппроксимации кривой, задаче о сумме подмножеств и задаче коммивояжёра

1. Обзор

1.1. Вероятностное программирование

Некоторые вероятностные языки программирования (ВЯП) расширяют существующие языки, сохраняя их семантику как частный случай. Программы на этих языках обычно включают вызовы (псевдо-) случайных функций. ВЯП используют расширенный набор случайных функций, соответствующих различным основным распределениям включая Гауссово, Бета, Гамма, полиномиальное, и т.д. Выполнение такой программы со случайным выбором представляется таким же образом как выполнение этой программы на основном (невероятностном) языке.

Однако программы в ВЯП рассматриваются как порождающие модели, определяющие распределения по возможным возвращаемым значениям, и их прямое вычисление может быть интерпретировано как взятие одной выборки от соответствующих дистрибутивов. Многократное вычисление программы может использоваться для оценки базового распределения.

ВЯП идут дальше и поддерживают программы определяющие условные распределения. Такие программы содержат заключительное условие, указывающее, должен ли результат оценки программы быть принят или нет. Самым простым способом выбора в условных распределениях является выборка с отклонением, в которой программа пересчитывается до тех пор пока ее заключительное условие не удовлетворено. Условные вероятности, определенные вероятностными программами, соответствуют апостериорным вероятностям порождающих моделей которым даны некоторые данные, таким образом их оценка очень полезна и может быть непосредственно применена к проблемам машинного обучения и вероятностному выводу. Однако выборка с отклонением может быть чрезвычайно неэффективной даже для довольно простых моделей.

Другой, более эффективный и широко используемый метод основывается на Цепях Маркова Монте-Карло, а именно, алгоритме Метрополиса-Гастингса.

Этот алгоритм использует стохастический локальный поиск, чтобы выбирать такие экземпляры, для которых данное условие останется истиной. Чтобы реализовать его для моделей, определенных вероятностными программами, нужно сделать небольшие изменения возвращаемых значений элементарных случайных процедур, вызывающихся в этих программах, и эти значения должны быть запомнены.

Алгоритм Метрополиса-Гастингса может быть намного более эффективным, чем выборка с отклонением для оценки апостериорных распределений. Однако не используя некоторые дополнительные методы он может быть настолько же плох как выборка с отклонением (или еще хуже из-за издержек) в получении первой надлежащей выборки. Можно легко проверить это на примере следующей простой программы на языке Church: (mh-query 1 1 (define xs (repeat 20 flip)) xs (all xs)).

В этой программе определен список из 20 случайных булевых значений, и этот список возвращается, когда все его значения истинны. Если заменить "mh-query 1 1" на "rejection-query" (выборку с отклонением), время вычисления немного уменьшится. Однако получение дальнейших выборок запросом mh-query будет намного быстрее, чем повторяющийся запрос с отклонением. Таким образом нерешенная проблема здесь - нахождение первого допустимого экземпляра модели. Это делается вслепую и в алгоритме Метрополиса-Гастингса и в выборке с отклонением.

Во многих практических проблемах пользователь может преобразовать строгое условие в мягкое или даже может первоначально иметь задачу с целью оптимизировать некоторую функцию.

1.2. Инструментарий

Поскольку исследования пространства решений в вероятностном программировании требуют манипуляций со случайным выбором, сделанным во время вычисления программы, разработки новых процедур запросов сопряжены с вмешательством в процесс вычисления. Так как никакой язык не поддерживает достаточно гибкое внешнее управление этим процессом, команде проекта было проще реализовать новый интерпретатор. Однако было решено не разрабатывать новый язык, а воспроизвести некоторую основную функциональность языка Church.

Базовым языком используется язык C++. Вместо функционального приложения, используются конструкторы классов. Например, такие функции как *define, lambda, list, cons, car, cdr, nullp, ListRef, if,* и др. реализованы в виде классов с соответствующими конструкторами. Все эти классы наследуются от класса *Expression*, у которого есть поле *std::vector<Expression*> children,* которое указывает на дочерние выражения. Таким образом, выражения составляют дерево.

Определены базовые математические операции (+, -, *, /), операции сравнения (<, <=, >, >=, =) и логические операторы (*not, and, or*). Реализован класс List, с соответствующими функциями работы со списками (*Car*, *Cdr*, *Cons*). Определены несколько функций распределения случайной величины (равномерное, Гауссово, полиномиальное и распределение Бернулли). Реализованы объявления переменных и функций (*define*, *let*) и вызовы функций с рекурсией.

Чтобы создать выражения из численных значений, добавлен класс *Value*. Этот класс используется для всех значений и динамически различает поддерживаемые типы.

Кроме того, были реализованы функции "*quote*" и "*eval*", которые соответственно выполняют цитирование выражения и вычисление значения процитированного выражения.

Таким образом, следующая запись программы на языке C++

Define(f, Lambda(xs, If(Nullp(xs), V(0), Car(xs) + f(Cdr(xs)))))

соответствует записи программы на языке Church

(define f (lambda (xs) (+ (if (null? xs) 0 (+ (car xs) (f (cdr xs)))))))

Чтобы использовать символы *f* и *xs*, нужно объявить их как экземпляры класса *Symbol* или записать *S ("xs")* вместо *xs*. Перегружен оператор применения параметров к функции, таким образом, можно записывать *f(xs)* вместо *Apply(f, xs),* где *Apply*, также дочерний элемент *Expression*. Точно так же можно записать *xs[n]* вместо *ListRef (xs, n).*

В интерпретаторе также обернуты некоторые функции и структуры данных библиотеки OpenCV, для удобства работы с графикой. Была добавлена поддержка матричного типа *cv::Mat* как основного, таким образом, можно записывать что-то типа *Define(S("image"), V(cv::imread("test.jpg"))). cv:: Mat* наследует все основные операции, таким образом значения, соответствующие *cv:: Mat* могут быть суммированы или умножены на другие значения.

Упомянутые конструкторы различных классов используются просто, чтобы создать выражения и расположить их в деревья. Во время выполнения, эти деревья выражений расширяются в трассировку программы. Трассировки программы - также деревья выражений, но в их узлах уже содержатся значения выражений.

Процесс выполнения и трассировки программы, реализованные в данной библиотеке C++, подобны реализованному в Church. Также, общий код C++ может легко использоваться вместе с данной вероятностной библиотекой программирования.

Данная библиотека разработана командой Aideus и используется в данной работе как инструмент исследования. Таким образом, мы больше не будем вдаваться в подробности.

1.3. Трассы выполнения программ

Трасса программы - дерево выражений, в узлах которого содержатся значения выражений. Каждое выражение в исходной программе, во время рекурсивного вычисления, преобразуется в структуру *(IR rnd? val expr)*, где *IR* - имя структуры, *rnd?* принимает значение *true*, если во время выполнения выражения *expr?* был сделан случайный выбор; *val* - результат вычисления (одна выборка от распределения, определенного *expr*). Для вычисления программы заданной в символьной форме, реализована функция *interpret-IR*. Рассмотрим некоторые примеры.

- *(interpret-IR '(10)) → (IR false 10 10)* значит что результат выполнения программы содержащей только одно выражение 10 это 10 и это не случайное значение.
- *(interpret-IR '(gaussian 0 1)) → (IR true -0.27 ('gaussian (IR false 0 0) (IR false 1 1)))* значит, что результат вычисления (gaussian 0 1) был -0.27 и это случайное значение, в то время как его параметры 0 и 1 были вычислены неслучайными значениями 0 и 1.
- *(interpret-IR '(if true 0 1)) → (IR false 0 ('if (IR false true true) (IR false 0 0) 1))* значит, что было вычислено только одно ответвление, в то время как другое сохранено в исходной форме.
- В более сложном случае случайное ответвление может быть расширено в зависимости от результата оценки стохастического условия: *(interpret-IR '((if (flip) 0 1))) → (IR true 1 ('if (IR true false '(flip)) 0 (IR false 1 1)))*. Здесь *(flip)* возвратило *false*, таким образом, второе ответвление было расширено. Также обратите внимание на то, что результат всего выражения случайный, в отличие от предыдущего примера.
- В определениях переменных в IR преобразовываются только их значения: *(interpret-IR '(define x (flip))) → 'define 'x (IR true false '(flip))*.

- Символы, которые могут быть найдены в окружающей среде, заменяются их значениями: *(interpret-IR '(define x (random-integer 10), x)) → ('define 'x (IR true 5 ('random-integer (IR false 10 10)) (IR true 5 'x))).*

Вычисленную программу можно рассматривать как расширенную первоначальную программу, но с дополнительной информацией о всем случайном выборе, сделанным во время вычисления. Эта программа может быть перевычислена снова, и ранее сделанный случайный выбор может быть принят во внимание. *Interpret-IR* может принимать и первоначальные программы и их IR расширения.

2. Генетическое программирование

Задача генетического программирования формализуется таким образом, чтобы её решение могло быть закодировано в виде вектора («генотипа») генов, где каждый ген может быть битом, числом или неким другим объектом

Некоторым случайным образом создаётся множество генотипов начальной популяции. Они оцениваются с использованием «функции приспособленности», в результате чего с каждым генотипом ассоциируется определённое значение («приспособленность»), которое определяет насколько хорошо фенотип, им описываемый, решает поставленную задачу.

При выборе «функции приспособленности» (fitness function) важно следить, чтобы её «рельеф» был «гладким».

Из полученного множества решений («поколения») с учётом значения «приспособленности» выбираются решения (обычно лучшие особи имеют большую вероятность быть выбранными), к которым применяются «генетические операторы» мутации и скрещивания, результатом чего является получение новых решений. Для них также вычисляется значение приспособленности, и затем производится отбор («селекция») лучших решений в следующее поколение.

Этот набор действий повторяется итеративно, так моделируется «эволюционный процесс», продолжающийся несколько жизненных циклов (поколений), пока не будет выполнен критерий остановки алгоритма.

Общую схему работы генетического алгоритма [6] можно представить следующим образом:

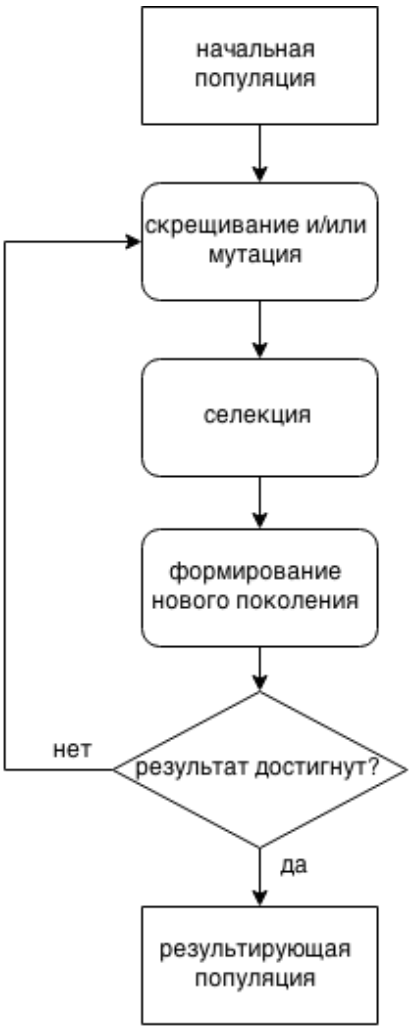

Рис. 1: Общая схема работы генетического алгоритма

Интерпретаторы вероятностных языков программирования предусматривают функции запросов (таких как mh-query), которые используются для вычисления апостериорных вероятностей или выполняют выборку в соответствии с указанным условием. Оптимизационные запросы основанные на генетическом программировании в данной работе принимают вместо строгих условий, функции, которые нужно минимизировать. Далее рассмотрим, как реализованы генетические операторы в этих параметрах настройки.

2.1. Оператор мутации

Для объединения генетического программирования с вероятностными языками, мы рассматриваем каждое выполнение программы как вариант решения. Источник вариации этих решений - различные результаты случайного выбора во время выполнении программы. Мутации состоят в небольших модификациях случайного выбора, выполненного во время предыдущего выполнения, которая напоминает некоторую часть алгоритма Метрополиса-Гастингса.

Во время каждого последующего перевычисления расширенной программы детерминированные выражения не пересчитываются, а используются их предыдущие значения. Все стохастические выражения высчитываются таким же образом как во время первого расчета кроме вызовов основных случайных функций, чье поведение изменено. Значения этих функций изменяются с учетом ранее возвращенных значений. Параметр скорости мутации *p* добавленный к *interpret-IR* указывает, как близко новые значения должны быть к предыдущим значениям. Например, предыдущий результат *(flip)* изменен с вероятностью, равной p. Переоценка *(IR true v (gaussian x0 s))*,где *v* - ранее возвращенное значение, *x0* - среднеквадратичное отклонение, и *s* - сигма, будет соответствовать *(gaussian v (* p s))*.

Например, результат переоценки IR-выражения *(IR true -0.27 ('gaussian (IR false 0 0) (IR false 1 1))* использующий *p=0.01*, может быть *(IR #t -0.26 ('gaussian (IR false 0 0) (IR false 1 1)))*.

2.2. Оператор скрещивания

Оператор скрещивания также использует трассировки программы. Он требует две переоценки расширений программы. Эти расширения интерпретируются вместе как одна и та же программа, структуры которых соответствуют (кроме изменений, вызванных случайным выбором). Основное различие находится в приложении основных случайных функций, так как должны быть приняты во внимание ранее возвращенные значения от обоих родителей.

Например, в моей реализации, два вызова *flip* в произвольном порядке возвращает одно из предыдущих значений и два вызова Гауссова распределения возвращает *(+ (* v1 e) (* v2 (-1 e)))*, где *v1* и *v2* - предыдущие значения, и *e* - случайное значение в [0, 1]. Мутации представлены одновременно с скрещивание для большей эффективности.

Для равномерного распределения я представил пять операторов скрещивания, которые выглядят следующим образом. Первый оператор возвращает случайным образом одно из значений родителей. Второй возвращает случайное значение находящееся между родителями. Третий использует дополнительные границы, и также возвращает случайное значение между родителями. Четвертый возвращает значение Гауссова распределения между родителями с дополнительными границами, и последний оператор сначала случайно выбирает одного из родителей, затем делает от него небольшое нормальное отклонение. На следующем изображении видно как работает каждый из операторов скрещивания.

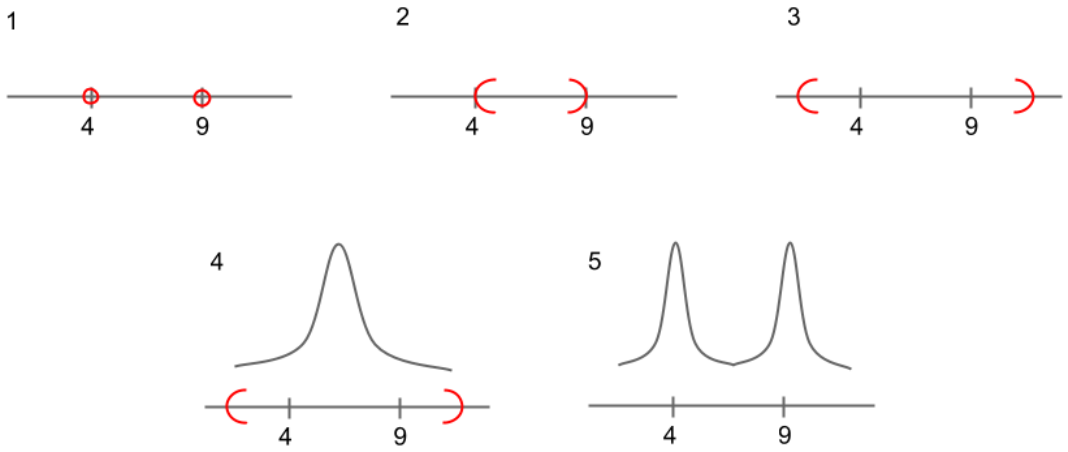

Рис. 2: Различные операторы скрещивания для равномерного распределения

Интересно, что дочерние элементы могут содержать ранее расширенный, но теперь неиспользуемые ответвления, которые могут быть активированы снова в более поздних поколениях из-за единственной мутации или скрещивания.

Могло бы казаться, что этот подход допускает простую форму генетического программирования только для параметрических программ, но это действительно не имеет место, так как на функциональных вероятностных языках можно легко программировать произвольные порождающие модели включая те, которые генерируют другие программы. Давайте рассмотрим некоторые простые примеры.

Рассмотрим два результата вычисления *'(gaussian 0 1)* и их скрещивание:

1. (list (IR #t 0.113 (list 'gaussian (IR #f 0 0) (IR #f 1 1))))
   (list (IR #t 0.447 (list 'gaussian (IR #f 0 0) (IR #f 1 1))))

   →

   (list (IR #t 0.236 (list 'gaussian (IR #f 0 0) (IR #f 1 1))))

Здесь видно, что результат скрещивания находится где-то между родителями.

2. (list (IR #t 9

(list 'if (IR #t #f '(flip)) -10

(IR #t 9 (list 'random-integer (IR #f 10 10))))))

(list (IR #t -10

(list 'if (IR #t #t '(flip)) (IR #f -10 -10)

'(random-integer 10))))

→

(list (IR #t 9

(list 'if (IR #t #f '(flip)) (IR #f -10 -10)

(IR #t 9 (list 'random-integer (IR #f 10 10))))))

Здесь были переоценены различные ответвления в родительских трассировках программы, и результат *(flip)* во время перекрестного соединения возвратил значение первого родителя. Так как случайное целое число еще не было оценено во втором родителе, его результат не перевычисляется, а берется непосредственно от первого родителя.

3. Оценка точности запросов

В следующих задачах сравниваются точности работы запросов использующих генетическое программирование с запросом использующего алгоритм Метрополиса-Гастингса в определении языка Church.

3.1. Задача аппроксимации кривой

Постановка задачи: найти коэффициенты полинома, которому принадлежат заданные точки.

Нужно заметить, что для того чтобы сравнивать точность работы алгоритма Метрополиса-Гастингса, использующий для выборки условие, с запросами генетического программирования, использующих минимизацию функции, для первого нужно определить функцию шума, чтобы время работы запросов было примерно одинаковым. Для этого нужно определить функцию *noisy-equals?* для mh-query:

```
(define (noisy-equals? x y)
(flip (exp (* -30 (expt (- x y) 2)))))
```

Результаты сравнения запросов с различными операторами скрещивания с запросом Метрополиса-Гастингса (М-Г) показаны в Табл. 1.

Данные	Среднеквадратичное отклонение					
	М-Г	случайно из родителей	равномерно между родителями	равномерно с доп. границами	нормально с доп. границами	нормальное отклонение от случ. родителя
$4x^2+3x$	1.710	0.221	0.198	0.140	0.042	0.035
$4x^2+3x$ друг. точки	0.940	0.434	0.177	0.112	0.026	0.010
$0.5x^3- x$	0.467	0.157	0.138	0.098	0.012	0.007

Табл.1: Сравнение методов на задаче аппроксимации кривой

3.2. Задача о сумме подмножеств

Постановка задачи: найти подмножество набора чисел, сумма элементов которого равна 0.

В нашем случае, запрос решающий эту задачу будет выглядеть следующим образом:

```
Symbol n, xs, ws, makews, sum, summ, xxs, wws;
sGPQuery(List() << include_libdefs()
        << Define(xs, List() << V(9568) << ... << V(5104) << V(1551))
        << Define(n, Length(xs))
        << Define(ws, ::repeat(n, Lambda0(Flip())))
        << Define(summ, Lambda(xxs, wws,
                If(Nullp(xxs), V(0.),
                        If(Car(wws), Car(xxs), V(0)) + summ(Cdr(xxs), Cdr(wws)) )
                        ))
        << Define(sum, summ(xs, ws))
        << ws
        << sum*sum
         )->reeval(&env, NULL, 1);
```

Для данной задачи были выбраны наборы из 20-25 чисел в диапазоне [-10000, 10000]. Алгоритм Метрополиса-Гастинга показал 85% правильных ответов, а запросы с использованием генетического программирования дали результат 75-80% правильных ответов. Также нужно отметить, что остальные 20-25% неправильных ответов были крайне близки к правильным: они нашли подмножество, сумма которого равна 1.

3.3. Задача коммивояжёра

Постановка задачи: найти кратчайший путь, проходящий через заданные точки только по одному разу.

Было использовано 2 набора точек: 6 и 30. Для алгоритма Метрополиса-Гастингса также как и в первой задаче используется функция шума *noisy-equals?*.

Данные	Среднеквадратичное отклонение					
	М-Г	случайно из родителей	равномерно между родителями	равномерно с доп. границами	нормально с доп. границами	нормальное отклонение от случ. родителя
6 точек	0.658	0.641	0.780	0.766	1.533	1.678
30точек	1.530	1.578	1.628	0.608	2.700	2.989

Табл. 2: Сравнение методов на задаче коммивояжёра

В данной задаче оператор скрещивания "случайно из родителей" показал результат близкий к алгоритму Метрополиса-Гастингса, а операторы "нормально с доп. границами" и "нормальное отклонение от случайного родителя" показали худшие результаты, в то время как на задаче аппроксимации кривой, они были лучшими и работали быстрее, чем М-Г.

Заключение

В результате работы, были разработаны запросы с использованием генетического программирования по трассировкам вероятностных программ. Результаты работы генетических операторов по трассировкам программ были сравнены с результатов работы алгоритма Метрополиса-Гастингса и показали различные результаты. На задачах аппроксимации кривой и коммивояжёра они показали результат не уступающий по точности алгоритму Метрополиса-Гастингса. Интересно объединить вероятностное программирование с усовершенствованными генетическими системами программирования, такими как MOSES [9].

Однако, эффективность общих методов вывода недостаточна, и это могло быть одним из принципиальных препятствий в пути к Общему Искусственному Интеллекту. Возможно, один общий метод вывода не может быть эффективным во всех проблемных областях, таким образом, он должен быть автоматически специализирован для каждого направления, с которым встречается AGI-агент. Это подразумевает, что такие методы должны быть глубоко объединены с познавательной архитектурой.

Список литературы

1. Batishcheva V., Potapov A. Genetic Programming on Program Traces as an Inference Engine for Probabilistic Languages // Lecture Notes in Artificial Intelligence. Springer, 2015.

2. Potapov A., Batishcheva V., Rodionov S. Optimization Framework with Minimum Description Length Principle for Probabilistic Programming // Lecture Notes in Artificial Intelligence. Springer, 2015.

3. Koller, D., McAllester, D.A., Pfeffer, A.: Effective Bayesian inference for stochastic programs. Proc. National Conference on Artificial Intelligence (AAAI), pp. 740–747 (1997).

4. Stuhlmüller, A., Goodman, N. D.: A dynamic programming algorithm for inference in recursive probabilistic programs. In: Second Statistical Relational AI workshop at UAI 2012 (StaRAI-12), arXiv:1206.3555 [cs.AI] (2012).

5. Milch, B., Russell, S.: General-purpose MCMC inference over relational structures. Proc. 22nd Conference on Uncertainty in Artificial Intelligence, pp. 349–358 (2006).

6. D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning." Addison- Wesley Publishing Co., Inc., Reading, Mass, (1989).

7. M. Srinivas and L. M. Patnaik, "Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms." IEEE Transactions on systems, man and cybernetic, 24(4), 656-667. (1994).

8. Solomonoff, R.: Algorithmic Probability, Heuristic Programming and AGI. In: Baum, E., Hutter, M., Kitzelmann, E. (Eds). Advances in Intelligent Systems Research, vol. 10 (proc. 3rd Conf. on Artificial General Intelligence), pp. 151–157 (2010).

9. Goertzel, B., Geisweiller, N., Pennachin, C., Ng, K.: Integrating Feature Selection into Program Learning. In: Kühnberger, K.-W., Rudolph, S., Wang, P. (Eds.): AGI'13, LNAI 7999, pp. 31–39 (2013).

10. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B., Church: a language for generative models. Proc. of Uncertainty in Artificial Intelligence, arXiv:1206.3255 [cs.PL] (2008).

11. Minka, T., Winn, J.M., Guiver, J.P., Knowles, D.: Infer.NET 2.4. Microsoft Research Camb., http://research.microsoft.com/infernet (2010).