Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

Остапенко Степан Сергеевич

Выпускная квалификационная работа

Проверка выполнимости SMT-формул с помощью нейронных сетей

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5156.2020
«Современное программирование»

Научный руководитель:
доцент,
Факультет математики и
компьютерных наук СПбГУ,
к.ф.-м.н. Шалымов Дмитрий Сергеевич

Рецензент:
ведущий инженер ключевых проектов,
ООО «Техкомпания Хуавэй»,
Ковальчук Сергей Валерьевич

Санкт-Петербург
2024 г.

Содержание

Введение

SMT-формулы

SMT-формулы (от англ. *Satisfiability Modulo Theories*) являются одним из наиболее важных объектов в области практических применений математической логики. Они позволяют сначала формальным образом записать разнообразные утверждения из различных предметных областей, а затем с помощью специальных вычислительных методов проверить логическую выполнимость (состоятельность) этих утверждений.

SMT-формула является некоторым расширением SAT-формулы, которое позволяет использовать не только логические переменные и связки с ними, а ещё и выражения с участием объектов из некоторого домена (например: целые числа, вещественные числа, битовые векторы, массивы) и различными операциями с ними (например: числовая арифметика, битовые операции, взятие элемента массива по индексу и т. д.).

Более формально, SMT-формула является формулой в логике первого порядка, где у каждой константы и у каждого символа переменной или функции есть некоторый заранее определенный тип (домен), а в качестве функциональных символов используются функции и операторы из разных доменов (например, те же битовые операции или взятия элемента массива по индексу). Задача проверки выполнимости[1] такой формулы состоит в том, чтобы выяснить, можно ли для каждой свободной (не находящейся под квантором) переменной подобрать значение соответствующего ей типа так, чтобы при подстановке данных значений формула была истинна. Помимо этого, чаще всего, если формула выполнима, требуется также для каждой свободной переменной выдать значение, сопоставляемое ей. Это отображение из множества свободных переменных в множество их значений называют моделью.

Чтобы стало более понятно, покажу несколько примеров.

Пример формулы:

$$(x + y + z = 2) \wedge (x + 2y + 3z = 2) \wedge (x + 2y + 4z = 1), \qquad (1)$$

[1] В тексте работы иногда будет появляться выражение «решение формулы», что в данном контексте будет являться синонимом выражения «проверка выполнимости формулы».

где x, y и z — целые числа. Эта формула является выполнимой, поскольку при подстановке значений $x \leftarrow 1$, $y \leftarrow 2$, $z \leftarrow -1$ формула превращается в истинное утверждение.

Ещё пример:

$$(x = 2y) \wedge (x = 2z + 1). \tag{2}$$

Если потребовать, чтобы x, y и z в этой формуле были целыми числами, то формула, очевидно, не будет выполнимой (т. к. в этом случае написанное здесь обозначает высказывание *x является чётным и x является нечётным*). Однако, если разрешить переменным быть вещественными (или хотя бы рациональными), формула станет выполнимой. Этот пример показывает, что свойство выполнимости зависит от типовых (доменных) ограничений, наложенных на переменные и функциональные символы.

Пример формулы с кванторами:

$$\exists\, x, y, z : (n \geqslant 0) \wedge (x > 0) \wedge (y > 0) \wedge (z > 0) \wedge (x^n + y^n = z^n), \tag{3}$$

где все переменные являются целыми числами. В этой формуле присутствует только одна свободная переменная — n, поэтому искать подходящее значение нужно только для неё. Нетрудно заметить, что нам подходят значения $n = 1$ и $n = 2$, поэтому формула является выполнимой (но если условие $n \geqslant 0$ заменить на $n \geqslant 3$, формула станет невыполнимой согласно Великой теореме Ферма).

Примером формулы с битовыми векторами является формула (4).

$$(x \geqslant \texttt{\#b01000000}) \wedge (x + (x \texttt{ shl}^2 \texttt{ \#b00000001}) + \texttt{\#b11111111} < \texttt{\#b00100000}), \tag{4}$$

где переменная x является битовым вектором размера 8 (фактически, это беззнаковое числом размером 1 байт). Аналогично, все константы[3] в формуле тоже являются битовыми векторами размера 8 и записываются в бинарном формате. Стоит также отметить, что каждый битовый вектор мож-

[2]Здесь `shl` — *shift left*: x `shl` y есть битовый сдвиг вектора x влево на y битов.

[3]Вообще константы не принято выносить как отдельную сущность в формулах, поскольку их можно считать функциями от нулевого количества аргументов и, соответственно, обозначать каждую из них своим функциональным символом из логики первого порядка. Но в этой работе для удобства я буду отделять константы, содержащиеся в формулах.

но однозначно отождествить с числом через его запись в двоичной системе счисления (например, таким образом происходит сравнение векторов на больше-меньше).

В указанном примере демонстрируется возможность использования переменных, функций и операторов из домена битовых векторов, а также то, что их семантика может отличаться от общепринятых семантик в других доменах. Так, например, все арифметические операции с битовыми векторами (в нашем случае, это два сложения) выполняются с учётом битового переполнения. Из этого, в частности, следует, что формула из примера является выполнимой, так как нам подходит значение $x = \texttt{\#b01100000}$ (или 96 в десятичной системе счисления):

- $x = \texttt{\#b01100000}$;
- $x \geqslant \texttt{\#b01000000}$ очевидно выполняется;
- $x\ \texttt{shl}\ \texttt{\#b00000001} = \texttt{\#b11000000}$;
- $x + (x\ \texttt{shl}\ \texttt{\#b00000001}) = \texttt{\#b01100000} + \texttt{\#b11000000} = \texttt{\#b00100000}$ из-за битового переполнения;
- $x + (x\ \texttt{shl}\ \texttt{\#b00000001}) + \texttt{\#b11111111} = \texttt{\#b00100000} + \texttt{\#b11111111} = \texttt{\#b00011111}$ опять же из-за переполнения;
- $x + (x\ \texttt{shl}\ \texttt{\#b00000001}) + \texttt{\#b11111111} = \texttt{\#b00011111} < \texttt{\#b00100000}$, что и требовалось.

SMT-логики

Говоря про домены, модальности и типы данных в SMT-формулах, важно рассказать про SMT-логики (в русскоязычной литературе их ещё часто называют теориями). Логики задают ограничения на типы переменных, констант, функций и операторов, которые можно использовать в формуле. Согласно языку логики первого порядка, в каждой формуле вне зависимости от теории можно использовать пропозициональные переменные (ложь, истина) и константы, логические связки между ними и кванторы. Возможность использования остальных конструкций уже определяется логикой.

Так, например, логика LIA (*Linear Integer Arithmetic*) позволяет использовать только целочисленные переменные и константы сколь угодно больших по модулю значений, а также арифметические операции и операторы сравнения (больше-меньше-равно) с их участием. Вдобавок, поскольку это линейная арифметика, все термы формулы обязаны быть линейными по каждой из переменных. Примером формулы в такой логике является формула (1).

Ещё один пример логики — LRA (*Linear Real Arithmetic*) — аналог LIA: устроена так же, но позволяет использовать только вещественные переменные и константы (т. е. проверить, решается ли уравнение в целых числах, с помощью этой логики не получится). Здесь уместно вспомнить формулу (2), которая является выполнимой в логике LRA, но не является выполнимой в логике LIA.

Нелинейные арифметики тоже реализованы в виде логик: NIA (*Non-linear Integer Arithmetic*) и NRA (*Non-linear Real Arithmetic*) во всём похожи на свои линейные аналоги, но не требуют линейности формулы по переменным (формула (3)).

Битовые векторы представлены в логике BV (*Bit Vector*). Эта логика позволяет записывать формулы с битовыми векторами произвольного фиксированного размера, используя битовые (и, или, исключающее или, отрицание, битовые сдвиги и т. д.), структурные (конкатенация двух битовых векторов, вырезание произвольного подотрезка битового вектора, циклические сдвиги и т. д.) и арифметические (сумма, разность, произведение, целочисленное деление, остаток от деления и т. д.) операции с ними, а также операторы сравнения.

Как уже было отмечено, все арифметические операции с битовыми векторами выполняются с учётом битового переполнения, а сравнения на больше-меньше осуществляются через сравнения целых чисел, двоичными записями которых являются сравниваемые векторы. Кроме того, для удобства, каждая арифметическая операция и каждое сравнение есть в двух вариантах: беззнаковом и знаковом. Знаковый вариант отличается тем, что двоичная запись числа (которая порождается битовым вектором) читается не в обычной арифметической семантике, где каждый бит обозначает очередную степень двойки, а в семантике «дополнение до двух», где самая старшая степень двой-

ки учитывается с минусом (именно таким образом хранятся знаковые целые числа в памяти современных компьютеров).

Если говорить про логики, применимые в компьютерных вычислениях, обязательно нужно упомянуть про логику FP (*Floating Point*), которая реализует вычисления с использованием чисел с плавающей точкой. Эта логика похожа на BV тем, что все переменные и константы представляют из себя битовые векторы соответствующих размеров (16, 32, 64, 128 битов), однако отличается тем, что при арифметических операциях и операциях сравнения данные битовые векторы воспринимаются не как двоичная запись целого числа, а как представление числа с плавающей точкой в памяти компьютера согласно стандарту IEEE-754.

Также важной частью являются логики, допускающие использование массивов и неинтерпретируемых функций. Такие логики имеют в названии часть A (*Array*) или UF (*Uninterpreted Function*) соответственно, например: ALIA, ABV, AFP, UFLIA, UFBV, UFFP или AUFBVFP.

Неинтерпретируемые функции работают как гомогенные отображения из объектов одного типа в объекты другого (возможно, того же самого), операцию применения которых можно использовать внутри SMT-формулы. Найти значение неинтерпретируемой функции — значит подобрать значение для каждого её аргумента так, чтобы записанные в формуле условия выполнялись.

Массивы работают аналогичным образом, однако, значения в них можно менять с помощью “присваивания по индексу” — специальной операции, которая создаёт новое отображение с изменённым значением у указанного аргумента.

Ещё особое место в списке логик занимают специальные безкванторные логики — это аналоги обычных логик, в которых запрещено использование любых кванторов существования и всеобщности. Такие логики обозначаются специальным префиксом QF_ (*Quantifier-Free*) в названии (например: QF_LIA, QF_NRA, QF_BV и т. д.).

Чтобы получить более сложные логики, можно объединить две уже существующие. В этом случае, множества допустимых к использованию в формуле объектов и операций объединятся. Например: при объединении логик

LIA и LRA получится логика LIRA, при объединении QF_BV и QF_LIA получится QF_BVLIA, при объединении LRA и QF_NRA получится NRA, а при объединении QF_ABV и QF_UFBV получится логика QF_AUFBV. Некоторые примеры расширения логик схематично изображены на рис. 1.

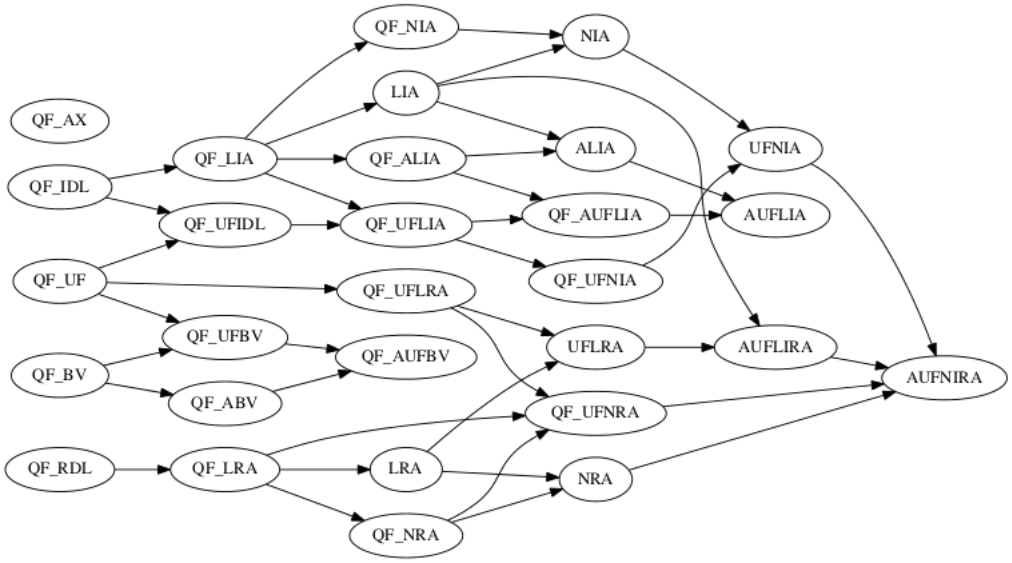

Рис. 1: Иерархия, состоящая из некоторых логик, поддерживаемых общепринятым стандартом SMT-LIB [1]. Если из логики L_1 проведена стрелка в логику L_2, то L_1 является частным случаем L_2.

Чтобы дополнить список примеров логик напомню, что пропозициональные переменные и константы, а также логические связки между ними по-умолчанию присутствуют в каждой логике. Поэтому существует логика BF (*Boolean Formula*), которая не содержит ничего кроме них, и логика QF_BF — её безкванторный аналог. Нетрудно заметить, что решение формул в логике QF_BF эквивалентно решению задачи SAT.

SMT-решатели

Несмотря на то, что даже самая простая логика QF_BF уже является NP-полной (т. е. задача решения формул в такой логике является NP-полной), а некоторые логики (например, NIA) являются неразрешимыми, люди всё равно изобретают различные эвристические алгоритмы и прочие подходы к решению задачи проверки выполнимости формул в таких логиках. Комбинации из таких подходов реализуются в специальных программах, называемых

SMT-решателями[4].

Одним из наиболее известных и, по совместительству, одним из первых из промышленных (т. е. активно используемых как составная часть более сложных программ) SMT-решателей является Z3 [2], разработанный в Microsoft Research. Он умеет выполнять проверку выполнимости формул и поиск модели, поддерживая самые разнообразные логики.

У Z3 есть конкуренты: решатель cvc5 [3], разработанный в Стэнфордском университете и Университете Айовы, и решатель Yices2 [4], разработанный в исследовательском центре SRI International (Stanford Research Institute). Упомянутые решатели регулярно побеждают Z3 на SMT-COMP [6] [7], проводимом ежегодно соревновании по решению SMT-формул, но на практике работают не так стабильно, поэтому реже используются в реальных прикладных задачах.

Ещё одним интересным примером является SMT-решатель Bitwuzla [5], который заточен специально под решение формул с битовыми векторами и без кванторов, поскольку многие практические задачи (например, символьное исполнение) формулируются как раз в таком виде.

Поскольку задача проверки выполнимости формулы является вычислительно сложной, и её решение в большинстве случаев полностью опирается на эвристики, зачастую SMT-решатели не могут выдать никакой ответ для той или иной формулы, даже спустя несколько дней работы. Это может стать существенным препятствием для решения задачи из реальной жизни, которая опирается на подобные формальные методы.

Применения

SMT-формулы обладают достаточно сильной выразительной способностью, поэтому решение многих прикладных задач можно свести к проверке выполнимости и поиску решения для SMT-формулы в некоторой логике.

Среди типовых применений можно выделить планирование задач и составление расписания, вывод типов в языках программирования с зависимыми типами и средства формального вычислительного доказательства теорем,

[4]Или SMT-солверами — англицизм, также используемый в русском языке.

верификация программного и аппаратного обеспечения, статический анализ программ, генерация тестового набора и символьное исполнение. Последнее рассмотрим более подробно.

Символьное исполнение

Символьное исполнение — техника интерпретации программных инструкций, при которой вместо исполнения программы одним конкретным путём, полностью определяемого данными, отправленными программе на вход (т. е. вместо обычного последовательного интерпретирования программных инструкций), программа исполняется сразу всеми возможными путями (при всех возможных комбинациях подаваемых на вход данных). Такой подход был впервые предложен в 76-м году в статье [8].

Проще всего думать про этот процесс так, как будто интерпретатор перебирает все варианты входных данных и запускает программу на каждом из них, анализируя её поведение. В действительности же такой подход почти невозможно реализовать из-за экспоненциального роста количества вычислений при росте размера входных данных. Поэтому на практике используют подход с неопределёнными входными данными.

Это выглядит примерно так: в процессе исполнения есть набор состояний, каждое из которых соответствует выполнению скольких-то первых инструкций программы на некоторых входных данных; у каждого состояния есть ограничения пути (*path constraints*) — SMT-формула, которая задаёт условия, которым должны удовлетворять входные данные, чтобы после выполнения определённого количества инструкций программа оказалась в таком состоянии (т. е. на той же инструкции и с такой же памятью). Если очередная инструкция не является ветвлением, то следующее состояние вычисляется однозначно. Если нет, то создаётся два новых состояния, одно из которых соответствует выполнению условия ветвления, а второе — невыполнению (у обоих состояний добавляются соответствующие ограничения пути).

Чтобы понять, существуют ли входные данные, приводящие программу в такое состояние, достаточно проверить на выполнимость её формулу ограничений пути. Здесь на помощь приходит SMT-решатель, который может сделать это и в процессе отсечь сразу часть недостижимых состояний, чьи

ограничения пути не выполняются (т. е. для которых не существует входных данных, которые приводят программу в это состояние).

Такой подход позволяет относительно эффективно искать различные пути исполнения программы и исследовать их: собирать ограничения на входные данные и их инварианты, которые сохраняются в процессе очередного исполнения. В дальнейшем собранные данные можно использовать для задач статического анализа программ, автоматической генерации тестового покрытия, поиска ошибок и уязвимостей в программах.

Одним из популярных инструментов, реализующих такой подход, является символьный движок KLEE [9], который осуществляет символьное исполнение программ, представленных в виде LLVM-байткода, и используется для генерации тестов и поиска уязвимостей в программном обеспечении [10].

Здесь же нужно отметить ещё один инструмент — символьный движок USVM [11], который исполняет программы, написанные на некотором специальном промежуточном языке X, что позволяет анализировать программы, написанные на любом языке, если для него есть транслятор в язык X.

Одно из возможных практических применений данной работы заключается как раз в оптимизации процесса символьного исполнения в USVM за счёт возможности быстро генерировать предсказания выполнимости для формул, задающих ограничения пути (*path constraints*). Такой подход может заранее отсеять многие состояния символьного исполнения, которым соответствует невыполнимое ограничение пути, и существенно ускорить весь процесс.

Ещё важное замечание: набор логик, в которых записываются SMT-формулы во время символьного исполнения, сильно ограничен. Это связано с тем, что память современных компьютеров ограничена и полностью дискретна, а значит для представления всех производимых над ней действий можно обойтись набором битовых векторов заранее известной длины. Таким образом, в процессе символьного исполнения возникают только формулы из логик `QF_BV`, `QF_ABV`, `QF_ABVFP`, `QF_AUFBV`, `QF_AUFBVFP`, `QF_BVFP`, `QF_FP`, `QF_UF`, `QF_UFBV` и `QF_UFFP`. Это в значительной степени помогает при построении SMT-решателей, предназначенных для процедуры символьного исполнения.

Постановка задачи

Поскольку, как уже было отмечено, SMT-решатели периодически зависают в процессе проверки очередной формулы, что негативно сказывается на времени решения прикладной задачи, хотелось бы иметь инструмент, который за достаточно разумное время мог бы посмотреть на формулу и гарантированно что-нибудь сказать про неё, например, выдать свою степень уверенности в том, что формула, на самом деле, является выполнимой.

Такой подход может быть применён на практике в случаях, когда в задаче есть сразу несколько формул, которые нужно проверить на выполнимость, и нам, в соответствии с некоторым здравым смыслом, интересно проверять только выполнимые формулы, или наоборот, только невыполнимые. Например, как уже сказано выше, в процессе символьного исполнения выполнимость формулы ограничений пути до состояния является критерием достижимости данного состояния. Каждый раз, когда мы находим достижимое состояние, система совершает прогресс[5], поэтому хочется как можно чаще проверять выполнимость ограничений пути в тех случаях, когда они действительно выполнимы.

В наше время задачи предсказания различных значений для сложно устроенных данных чаще всего решаются с помощью нейронных сетей. Таким образом, цель данной работы: *исследовать возможность применения нейронных сетей для предсказания выполнимости SMT-формул в различных логиках*.

Отмечу также, что в поставленной формулировке задачу можно рассматривать, в том числе, как задачу ранжирования, однако для большей общности, а также с учётом того, что в открытых источниках нет никакой информации об исследованиях про способы анализа выполнимости SMT-формул исключительно с помощью нейронных сетей, я буду пытаться найти решение в более простом виде — в виде решения задачи классификации. Тем не менее, одно из практических применений полученных моделей будет опираться на их использование в контексте ранжирования.

Для достижения поставленной цели я выделил следующие шаги:

[5]На самом деле, когда мы понимаем, что некоторое состояние недостижимо, система тоже совершает прогресс, но в большинстве случаев он является менее значительным.

1. Подобрать датасет[6] для обучения модели и оценки качества, а также соответствующие метрики.

2. Исследовать существующие подходы к представлению SMT-формул в различных задачах.

3. Исследовать различные архитектуры нейронных сетей, которые могут быть применены для решения общей задачи.

4. Реализовать инфраструктуру для подготовки данных, обучения моделей и оценки их качества.

5. Выполнить обучение моделей и оценить их качество.

6. * Если качество будет удовлетворительным, можно попытаться подготовить версию модели для использования на практике[7].

7. * Если и предыдущий шаг удастся сделать, можно будет попробовать встроить модель в упомянутый выше символьный движок USVM [11] и оценить прирост производительности в процессе символьного исполнения.

[6] Набор данных.

[7] Достижение этого шага заранее не предполагается, но если его получится выполнить, будет хорошо.

Обзор литературы и предметной области

Тема применения нейронных сетей в контексте решения формально-логических задач развита не особо сильно, поскольку дискретные задачи со строгими ограничениями тяжело даются сетям с известными на данный момент архитектурами. Тем не менее, несмотря на полное отсутствие подходов, аналогичных рассматриваемому в данной работе, и основанных на применении нейросетей для проверки выполнимости формул в логике первого порядка, известны многочисленные попытки использовать нейросети в качестве помощника для алгоритма, решающего ту или иную логическую, комбинаторную или оптимизационную задачу. Про несколько таких подходов, имеющих отношение к проверке выполнимости и поиску решения для SMT-формул, я расскажу далее. Но перед этим я детально опишу устройство и архитектуру графовых нейронных сетей, поскольку на использовании данной архитектуры в значительной части основана моя работа.

Устройство GNN

Графовые нейронные сети[8], наряду со свёрточными, рекуррентными и т. д., являются одним из способов решать задачу машинного обучения, учитывая специфику данных. Так как графы отображают взаимоотношения между объектами некоторого множества, то и процесс построения модели, в данном случае, учитывает локальность и связи между произвольными объектами из предметной области.

Впервые архитектура GNN в общем виде была описана ещё в 2009 году в статье [12], однако широкую известность получила только в 2017 году после её применения для предсказания квантовых чисел в вычислительной органической химии [13]. В основе вычислений лежит процесс *передачи сообщений* между вершинами графа, который изображён[9] на рис. 2: в каждой вершине хранится вектор-состояние (эмбеддинг[10]), и на каждом шаге все вершины

[8] Далее по тексту будем называть их GNN (англ. *Graph Neural Networks*).

[9] Подобное вычисление происходит для каждой вершины на каждой итерации процесса *передачи сообщений*.

[10] От англ. *embedding* — в машинном обучении так называют представление некоторого объекта в многомерном векторном пространстве. Далее по тексту слова “вектор”, “состояние” и “эмбеддинг” в контексте графовых нейросетей будут использоваться взаимозаменяемо.

сначала рассылают свой текущий вектор всем своим непосредственным соседям, а потом агрегируют пришедшие от соседей векторы (*сообщения*) и обновляют свой вектор-состояние с учётом этого; далее эти шаги повторяются несколько раз, после чего полученные таким образом векторы-состояния используются для построения представления графа и дальнейшего решения задачи.

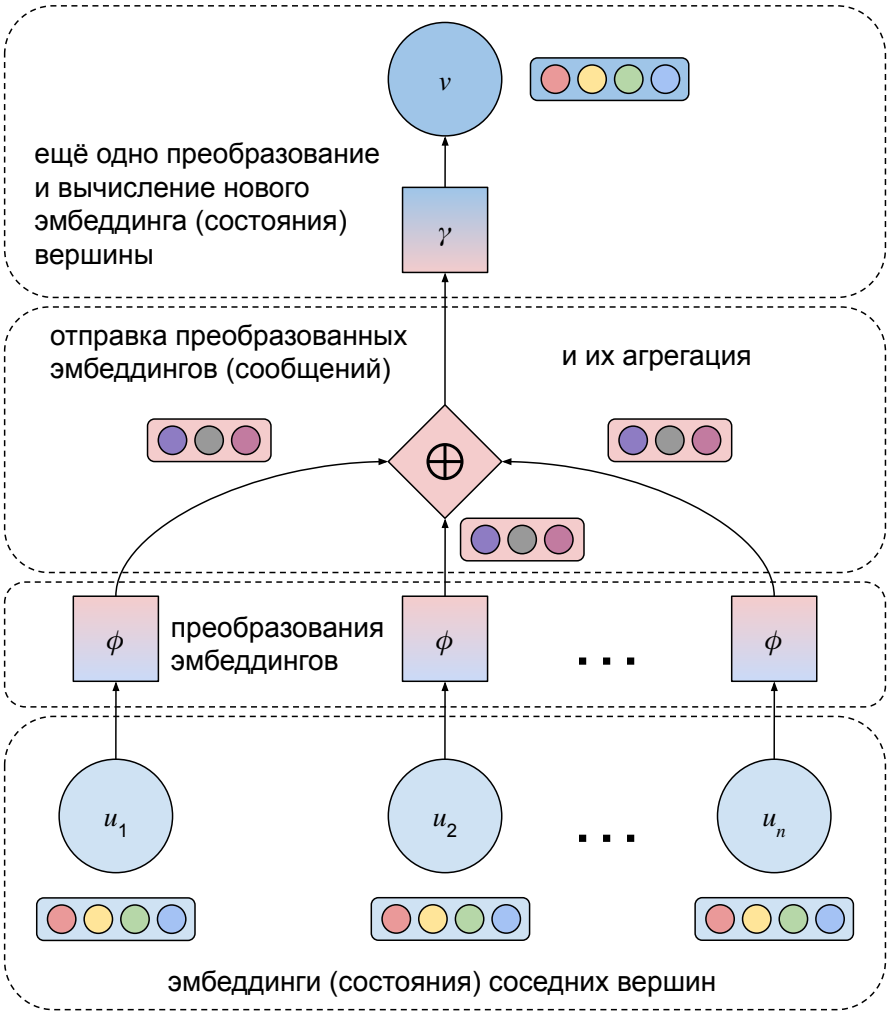

Рис. 2: Схема обновления вектора-состояния вершины v через векторы-сообщения от её соседей $u_1, u_2, \ldots, u_n$. Цвет (синий / красный) обозначает размерность. Более тёмный цвет вверху обозначает обновлённое состояние.

Согласно [14], формально весь процесс устроен следующим образом:

- у каждого ребра e есть набор параметров $s_e \in \mathbb{R}^m$;

- в начале вычислений в каждой вершине v содержится содержится вектор (состояние) $x_v^{(0)} \in \mathbb{R}^n$ с некоторой информацией;

- производится несколько итераций *передачи сообщений*, на t-й из них производится обновление векторов в вершинах по правилу, которое описывается формулой (5);

- после всех итераций предполагается, что вычисленные векторы (состояния) образуют некоторое представление вершины / графа, поэтому их можно использовать в качестве параметров[11] вершины / графа, подавая в какую-нибудь MLP[12]-сеть.

$$x_v^{(t)} = \gamma^{(t)} \left(x_v^{(t-1)}, \bigoplus_{u \in \mathcal{N}(v)} \phi^{(t)} \left(x_v^{(t-1)}, x_u^{(t-1)}, s_{e(u \to v)} \right) \right) \tag{5}$$

Обозначения в формуле (5):

- $x_v^{(t)} \in \mathbb{R}^n$ — описанные выше эмбеддинги вершин;

- $e(u \to v)$ — ребро из вершины u в вершину v, а $s_{e(u \to v)} \in \mathbb{R}^m$ — его параметры;

- $\mathcal{N}(v)$ — окрестность вершины v;

- $\phi^{(t)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^k$ — функция создания сообщения по состояниям вершин и параметрам ребра; может быть представлена аффинным преобразованием, композицией линейных слоёв и нелинейных активаций, какими-нибудь сложными LSTM[13] или GRU[14]-слоями [15] или просто любым дифференцируемым преобразованием с обучаемыми или необучаемыми параметрами;

[11] Будем также называть их признаками или фичами (от англ. *feature*).

[12] Многослойный перцептрон (от англ. *Multi-Layer Perceptron*) — последовательная комбинация из полносвязных линейных слоёв и нелинейных слоёв активации.

[13] Долгая краткосрочная память (от англ. *Long Short-Term Memory*) — известная модификация рекуррентной нейронной сети.

[14] Управляемый рекуррентный блок (от англ. *Gated Recurrent Unit*) — ещё одна известная модификация рекуррентной нейронной сети.

- $\bigoplus : \mathbb{R}^k \to \mathbb{R}^k$ — функция агрегации, например: сумма, среднее или взвешенная сумма с учётом механизма внимания;
- $\gamma^{(t)} : \mathbb{R}^n \times \mathbb{R}^k \to \mathbb{R}^n$ — функция обновления вектора-состояния (эмбеддинга) вершины; аналогична $\phi^{(t)}$.

В качестве примера построения сети по такой схеме можно рассмотреть Graph Convolutional Network [16], самую популярную GNN-архитектуру. В её случае мы имеем:

- $\phi^{(t)} = W^T \cdot x_u^{(t-1)}$;
- $\bigoplus = \sum\limits_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{\sqrt{|\mathcal{N}(v)|} \cdot \sqrt{|\mathcal{N}(u)|}} \cdot \phi^{(t)} \left(x_v^{(t-1)}, x_u^{(t-1)}, s_{e(u \to v)} \right)$;
- $\gamma^{(t)} = \bigoplus\limits_{u \in \mathcal{N}(v)} \phi^{(t)} \left(x_v^{(t-1)}, x_u^{(t-1)}, s_{e(u \to v)} \right) + b$;

где матрица весов W и вектор-сдвиг b являются выучиваемыми параметрами. Итого получаем:

$$x_v^{(t)} = \sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{\sqrt{|\mathcal{N}(v)|} \cdot \sqrt{|\mathcal{N}(u)|}} \cdot \left(W^T \cdot x_u^{(t-1)} \right) + b$$

В настоящий момент GNN широко применяются для решения задач вычислительной физики, химии и биологии, построения рекомендательных систем, прогнозирования трафика на дорогах, распознавания объектов, синтеза текстов и речи [17] [14]. Замечу, что, в некотором смысле, про эту архитектуру можно говорить как про результат скрещивания свёрточных и рекуррентных нейронных сетей.

FastSMT

В статье [18] рассматривается идея повышения эффективности SMT-решателя за счёт оптимизации его работы на формулах, возникающих в задачах из фиксированной предметной области.

Решатель в процессе своей работы применяет к формуле большое количество разных семантически эквивалентных преобразований[15], чтобы привести её к некоторому удобному для себя виду, в котором он сможет либо достаточно быстро найти решение для формулы и доказать, что оно подходит (тем самым доказав, что формула является выполнимой), либо достаточно быстро доказать, что указанный в формуле набор ограничений невозможно выполнить, и формула является невыполнимой. Примеры тактик: замена переменной, нормализация границ неравенства, bit-blasting (представление переменной в виде набора пропозициональных значений), переписывание условного оператора через конъюнкцию импликаций и т. д.. Упомянутый ранее решатель Z3 [2] поддерживает более ста подобных операций.

Цепочка преобразований, которые решатель производит над формулой, формируется согласно некоторой стратегии, которая учитывает самые разнообразные параметры (признаки) формулы: от её размера и количества свободных переменных до некоторого приближения абстрактно-синтаксического дерева формулы. Сама стратегия, в данном случае, больше всего похожа на решающее дерево, у которого в вершинах стоят ограничения на очередные параметры формулы, а в каждом листе находится тактика, которую стоит применить в случае, когда параметры соответствуют пути в этот лист.

Базовую стратегию, которая используется в решателе по-умолчанию, подбирают его создатели, причём они делают это так, чтобы его средняя скорость работы на произвольной формуле была как можно лучше. В результате, решатель работает более стабильно (т. е. реже уходит в экспоненциальный перебор возможных решений, когда на вход подают сложную формулу), но скорость нахождения ответа в более простых случаях из-за этого проседает.

В то же время, большинство популярных современных инструментов для решения SMT-формул поддерживают возможность использования специфических стратегий, которые можно построить, в том числе, самостоятельно. Помимо этого, есть некоторая интуиция, что если мы будем рассматривать только формулы из некоторой фиксированной практической задачи, то все эти формулы будут обладать некоторой спецификой, знание о которой сможет существенно помочь в проверке их выполнимости. На этой интуиции,

[15] В статьях и документациях их называют тактиками.

а также на возможности использовать собственные стратегии при решении формул строится подход, описанный в рассматриваемой статье.

Главная идея состоит в следующем: набор признаков формулы можно рассматривать как состояние некоторой среды, применение очередной тактики можно рассматривать как некоторое действие, совершаемое в этой среде, а награду за последовательность применённых тактик можно определить как суммарное время работы решателя на заданной формуле с использованием этой последовательности (разумеется, взятое с минусом), либо какому-нибудь штрафному значению в случае, если решатель не справляется с формулой. Таким образом, задача поиска оптимальной стратегии применения различных тактик хорошо представляется в виде задачи обучения с подкреплением, где средой является произвольная формула из некоторой фиксированной задачи. Поэтому предлагается выучивать модель (политику), которая по текущему состоянию среды (формулы) будет выдавать оптимальную тактику и параметры для неё.

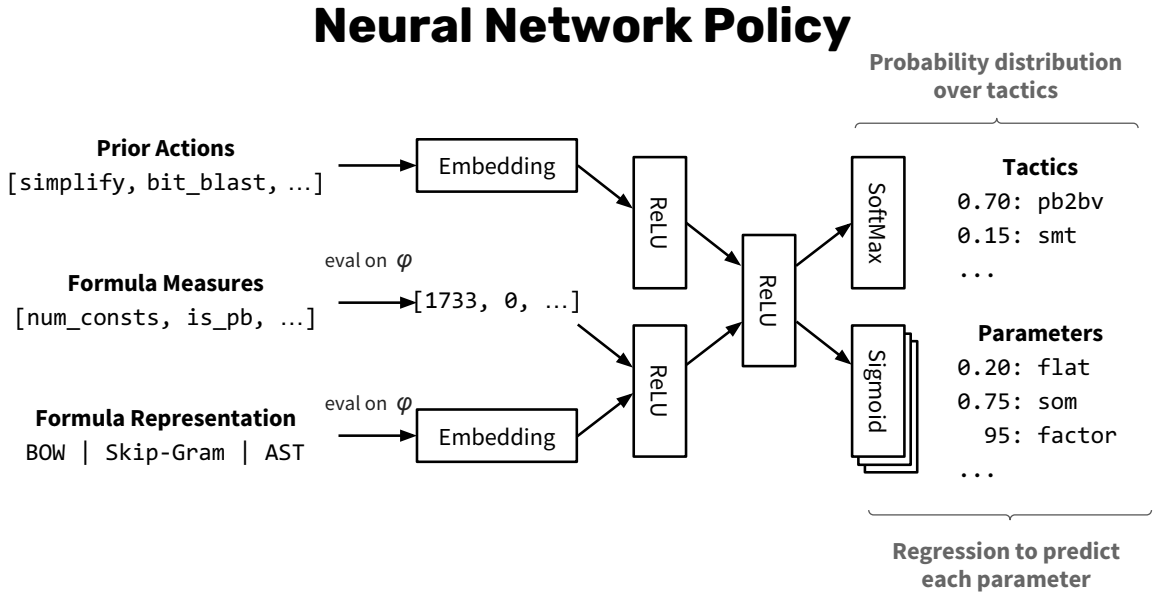

Рис. 3: Архитектура модели. Картинка взята из статьи [18].

Архитектура модели изображена на рис. 3. На вход подаются признаки формулы (Formula Measures) и её синтаксическое представление, закодиро-

ванное в виде bag-of-words или n-граммной модели (Formula Representation). Вдобавок к этому, для каждой тактики выучивается эмбеддинг, который хранит в себе семантическую информацию про неё. Эмбеддинги тактик, которые можно применить в данный момент, также подаются на вход модели (Prior Actions). На выходе у модели распределение на тактиках, которые стоит применить в данный момент, и значения параметров для них. Сразу отмечу, что, несмотря на то, что модель выдаёт распределение на всех доступных для применения в данный момент тактиках, за одно действие применяется только одна из них — наиболее вероятная.

Построенная таким образом модель обучается на наборе формул взятом из некоторой задачи (в статье это были различные известные бенчмарки для решателей, на каждом из которых обучалась и оценивалась отдельная модель) с помощью метода, похожего на кросс-энтропийный метод. После этого с помощью техник семплирования строятся разнообразные статистики, отражающие выученную моделью политику, а уже на них обучается решающее дерево, выбирающее нужную тактику по текущей формуле, которое впоследствии превращается в стратегию, которую можно загрузить в SMT-решатель.

Авторы статьи заявляют, что, согласно проведённым экспериментам, предложенный подход позволяет успешно решать на 17% больше формул при десятисекундном ограничении по времени, а также даёт прирост производительности вплоть до стократного ускорения на некоторых формулах (при этом, замедления при решении других формул не наблюдается). Тем не менее, у такого подхода есть существенный недостаток: для каждой новой задачи нужно сначала самостоятельно собирать данные, потом запускать тяжеловесный процесс обучения, а после этого самому синтезировать стратегию. Это требует довольно больших вычислительных мощностей.

GNN for Scheduling of SMT Solvers

В статье [19] рассматривается более общий подход к задаче: известно, что в основе работы разных SMT-решателей лежат разные алгоритмы и эвристики, поэтому их производительность при решении разных формул может значительно варьироваться; в связи с этим, давайте просто обучим модель, которая по формуле будет предсказывать, за какое время тот или иной реша-

тель сможет проверить её на выполнимость, а далее перед каждой проверкой будем выбирать решатель, для которого модель предсказывает наименьшее время работы.

Данная статья интересна тем, что в ней рассматривается более точное представление формулы, подаваемой модели на вход. Вместо сбора разных статистик с формулы и превращения их в признаки авторы предлагают использовать графовую нейронную сеть (GNN), которая задействует абстрактно-синтаксическое дерево[16] формулы в качестве графа вычислений (рис. 4).

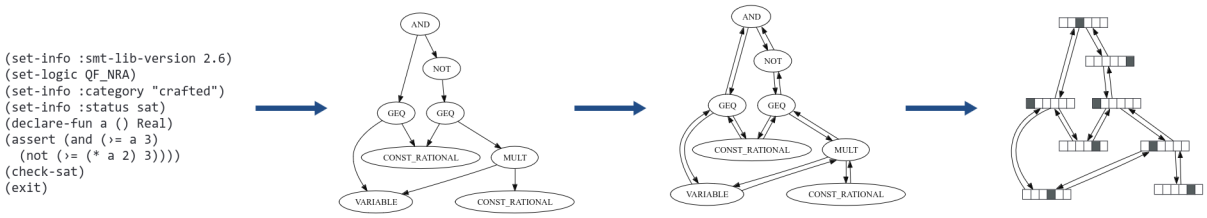

Рис. 4: Использование AST формулы в качестве графа вычислений для GNN. Картинка взята из статьи [19].

Чтобы ускорить работу, вершины дерева, отвечающие за эквивалентные выражения, склеиваются в одну, так что формула, на самом деле, превращается в ориентированный ациклический граф. Помимо этого, чтобы расширить распространение эмбеддинга с информацией, сохранённой в каждой вершине, к каждому ребру такого графа добавляется обратное ребро. Таким образом, информация начинает распространяться сразу во всех направлениях.

Далее каждая вершина (переменная, константа или операция) кодируется с помощью техники one-hot-encoding, после чего векторы с полученными таким образом значениями отправляются в GNN, на выходе у которой находится слой, собирающий эмбеддинги со всех вершин графа и предсказывающий для каждого SMT-решателя время, за которое он может выдать ответ на данной формуле. Для лучшего понимания, схема процесса изображена на рис 5.

[16] Далее в тексте будет использоваться аббревиатура AST (англ. *Abstract-Syntax Tree*).

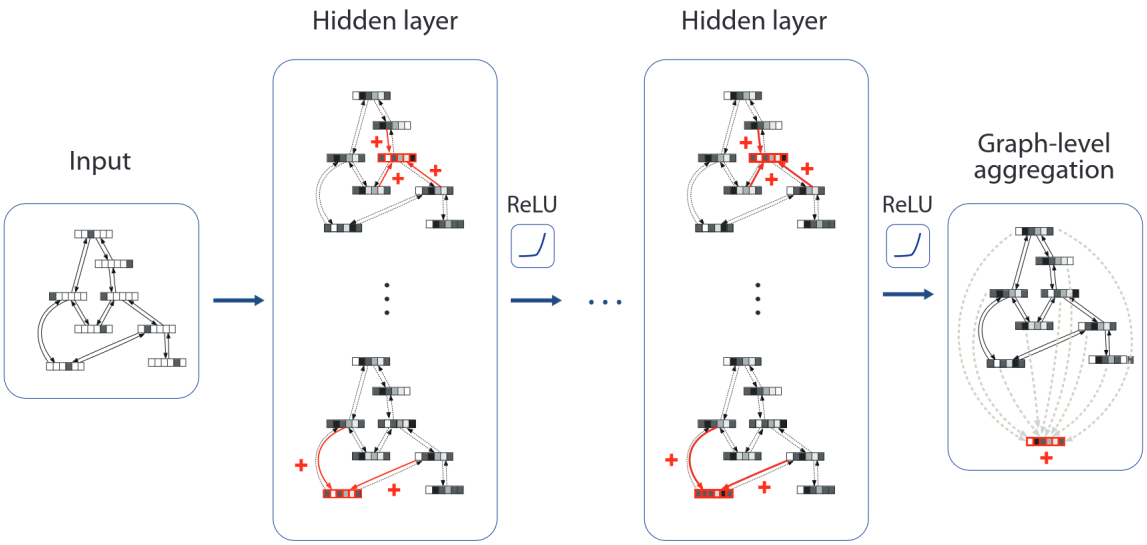

Рис. 5: Схема использования GNN для получения информации о формуле и предсказания ответа. Картинка взята из статьи [19].

Авторы отмечают, что подход может быть применён для решения почти для любой SMT-формулы, и утверждают, что им удалось добиться ускорения процесса на 8–900% на разных тестах.

NeuroSAT

Тем не менее, попытки научиться решать формально-логическую задачу, пользуясь исключительно нейронными сетями, тоже присутствуют. Пионером в этой области стала работа [20], в которой исследовалось применение GNN для задачи SAT. Была предложена следующая графовая архитектура: для каждой переменной, для отрицания каждой переменной и для каждого конъюнкта формулы заводится по вершине; далее рёбра проводятся между парами из литерала (переменными или их отрицаниями) и конъюнкта, если литерал входит в состав конъюнкта, а также между парами противоположных литералов (теми, которые является отрицаниями друг друга). Пример построения такого графа изображён на рис. 6.

Начальные состояния каждой вершины выучиваются и сохраняются в таблицу эмбеддингов. Затем во время вычислений каждая вершина накапливает все пришедшие к ней сообщения с помощью LSTM-слоя. После каждого шага передачи сообщений по графу каждая вершина-литерал генерирует соб-

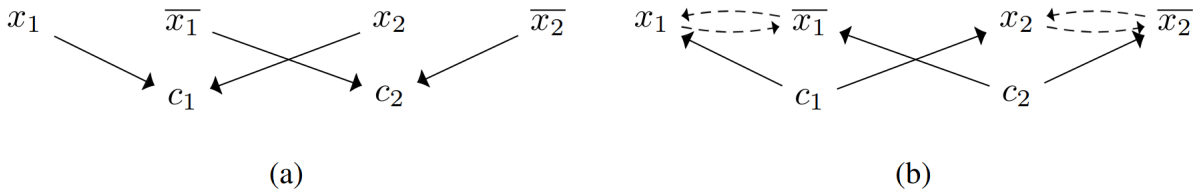

Рис. 6: Граф, построенный для формулы $(x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$. Пункты (a) и (b) отражают шаги построения, описанные в статье; на деле же граф выглядит как их объединение. Картинка взята из статьи [20].

ственное предсказание по поводу того, является ли данная формула выполнимой (внутреннее состояние LSTM отправляется в многослойный классификатор (MLP), который предсказывает одно число — уровень уверенности вершины в том, что формула выполнима), и все эти предсказания усредняются: из них получается одно число — общий уровень уверенности в том, что формула, по которой построен этот граф, выполнима.

Авторы статьи утверждают, что в течение первых скольких-то шагов передачи сообщений вершины будут “голосовать” хаотично, не ориентируясь на чужие предсказания, но в один момент произойдёт переход, и все вершины внезапно станут выдавать одинаковый ответ (например, все станут голосовать за то, что формула выполнима), причём эта тенденция никогда не нарушится, сколько бы шагов передачи сообщения после этого момента не последовало. Пример такого процесса изображён на рис. 7: сначала вершины не уверены в том, какой, на самом деле, является формула, но в какой-то момент все резко начинают голосовать за то, что она выполнима.

Модель обучалась на датасете **SR**(40), содержащем случайные SAT-формулы с не более чем 40 переменными и конъюнктами размера 3–4. По заявлению авторов, если взять такую модель и провести валидацию на отложенной выборке из **SR**(40), каждый раз запуская 26 шагов передачи сообщения, модель наберёт 85% по метрике точности, а в 70% случаев даже сможет восстановить ответ (т. е. найти значения переменных, при которых формула истинна). Причём у неё наблюдается некоторая способность к обобщению на формулы большего размера: иногда модель справляется даже с формулами из **SR**(200) (см. рис. 8).

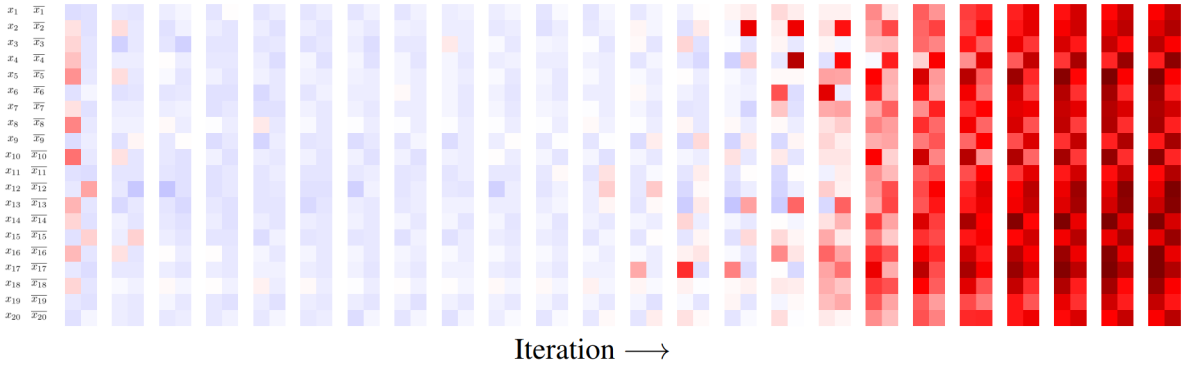

Рис. 7: Предсказания вершин графа по итерациям. В каждой строчке даны пары предсказаний для вершины-переменной и для её отрицания. Отдельные столбцы обозначают разные итерации процесса передачи сообщений. Цвета обозначают предсказания вершин: красный — уверенность в том, что формула выполнима, синий — наоборот. Интенсивность цвета обозначает уровень уверенности в своём предсказании: чем насыщеннее, тем больше уверенности. Картинка взята из статьи [20].

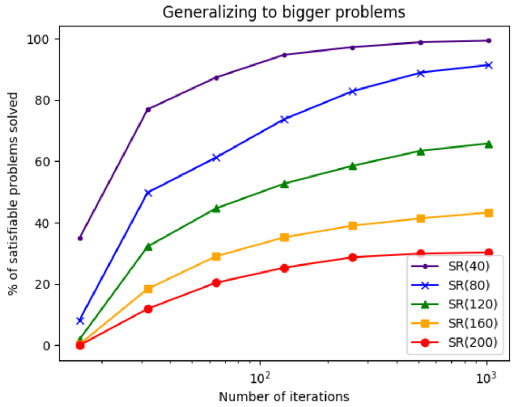

Рис. 8: Зависимость процента успешно решённых формул от количества шагов передачи сообщений для разных датасетов вида $\mathbf{SR}(n)$. Картинка взята из статьи [20].

В статье утверждается, что если изначально учить модель на формулах большего размера и делать больше шагов передачи сообщений, можно существенно улучшить качество.

1. Датасет

В своей работе я использовал два датасета с SMT-формулами:

1. Бенчмарк[17] с SMT-COMP 2023 [21].
2. Формулы, собранные в процессе работы дважды упомянутого выше символьного движка USVM [11].

1.1. SMT-COMP

В процессе исследований этот бенчмарк был поделен на три части:

1. `BitVec` — формулы из логики `QF_BV`, обычные формулы с битовыми векторами;
2. `SymbEx` — формулы из логик `QF_BV`, `QF_ABV`, `QF_ABVFP`, `QF_AUFBV`, `QF_AUFBVFP`, `QF_BVFP`, `QF_FP`, `QF_UF`, `QF_UFBV` и `QF_UFFP`, которые возникают в процессе работы любого движка для символьного исполнения;
3. `QuaFree` — формулы из всех безкванторных логик, включая `QF_LIA`, `QF_NRA`, `QF_BVFPLRA` и т. д..

Такое разделение обусловлено тем, что для анализа качества хочется смотреть, как модель обучается и какое качество она выдаёт на формулах из разных логик. Однако, к сожалению, большинство логик в данном бенчмарке содержат слишком мало данных, чтобы можно было провести анализ на них, поэтому было принято решение объединить логики в группы по смыслу. Логика `QF_BV` была вынесена в отдельную группу, поскольку она является самой многочисленной, а также самой важной на практике. Логики из группы `SymbEx` были собраны вместе, поскольку моя текущая задача существует в контексте символьного исполнения, поэтому хочется отдельно анализировать способность модели решать подобные формулы. В оставшуюся группу `QuaFree` попали все безкванторные логики, так как хочется также анализировать способность модели решать задачу в некотором общем случае. В

[17]Набор данных для тестирования корректности или производительности программы.

дальнейшем каждую группу будем называть датасетом с соответствующим именем.

Формулы с кванторами в данной работе не рассматривались вообще, поскольку они существенно сложнее безкванторных, и их было решено отложить до лучших времён.

Чтобы данные влезли на видеокарту в каждом датасете были оставлены только формулы размером[18] не более 10 000 и глубиной[19] не более 2 000. Итоговые параметры построенных датасетов отображены в таблице 1.

Датасет	Количество формул	Средний размер формулы	Средняя глубина формулы	Доля выполнимых формул
`BitVec`	33 797	1181.92	85.45	0.378
`SymbEx`	85 078	669.46	49.24	0.610
`QuaFree`	123 396	965.16	48.71	0.622

Таблица 1: Параметры датасетов, полученных из данных с SMT-COMP 2023 [21].

1.2. USVM

Для проверки возможности применения модели на практике были также собраны датасеты, состоящие из формул, которые возникают в процессе работы символьного движка USVM [11]. Такая идея возникла, поскольку в статье [18], описанной в разделе FastSMT, было показано, что обучение модели на формулах, собранных с какой-то конкретной задачи, может помочь в решении других формул, возникающих в этой же самой задаче.

Сбор осуществлялся с помощью простого логирования формул, на которых вызывался SMT-решатель. Разные датасеты получались при запуске движка на разных проектах или наборах программ. Подобным образом были собраны три тренировочных и восемнадцать валидационных датасетов.

Такое строгое разделение на тренировочные и валидационные датасеты

[18]Размером формулы считаем количество переменных, констант и операций в ней. Например, размер формулы $(x + y = 5) \vee (z = 3)$ равен 9.

[19]Глубиной формулы считаем максимальную вложенность переменных, констант и операций в ней. Например, глубина формулы $(x + y = 5) \vee (z = 3)$ равна 4.

вдобавок к такому большому количеству вторых обусловлено желанием проверить обобщающую способность модели: хочется, чтобы модель показывала хорошее качество на формулах, возникающих при запуске движка на любых программах; при этом, нет никаких гарантий, что при переходе от одного набора программ к другому распределение, из которого порождаются формулы, изменится несущественно, и качество модели, обученной на данных из другого распределения, упадёт не слишком сильно.

Тренировочные датасеты были собраны на следующих программах:

1. `usvm-test` — на наборе программ для unit и интеграционного тестирования USVM;
2. `the-algorithms` — на репозитории с реализациями различных теоретических и практических алгоритмов на Java;
3. `usvm-core` — на ядре символьного движка USVM (движок был запущен на самом себе).

Параметры тренировочных датасетов указаны в таблице 2.

Датасет	Количество формул	Средний размер формулы	Средняя глубина формулы	Доля выполнимых формул
`usvm-test`	153 778	522.84	5.18	0.038
`the-algorithms`	181 633	277.03	23.94	0.066
`usvm-core`	192 744	179.42	12.19	0.066

Таблица 2: Параметры тренировочных датасетов, собранных в процессе работы USVM.

Валидационные датасеты были собраны при запуске USVM на следующих проектах, написанных на Java и других JVM-языках:

1. `owasp` — OWASP[20], открытый бенчмарк из программ на Java, на котором оценивают качество инструментов для автоматического поиска уязвимостей в коде [23];
2. `cassandra` — Apache Cassandra, распределённая NoSQL СУБД[21] [24];

[20] Open Web Application Security Project.

[21] Система Управления Базами Данных.

3. `kafka` — Apache Kafka, распределённый брокер сообщений [25];
4. `spark-core` — Apache Spark, система для распределённой обработки данных (модуль ядра) [26];
5. `spark-streaming` — тот же Spark (модуль потоковой обработки данных);
6. `utbot-core` — UnitTestBot, инструмент для автоматической генерации unit-тестов (модуль ядра) [27];
7. `utbot-java` — тот же UnitTestBot (модуль генерации тестов для Java);
8. `utbot-python` — тот же UnitTestBot (модуль генерации тестов для Python);
9. `utbot-js` — тот же UnitTestBot (модуль генерации тестов для языка JavaScript);
10. `utbot-go` — тот же UnitTestBot (модуль генерации тестов для Go);
11. `zookeeper` — Apache Zookeeper, служба для координации распределённых систем [28];
12. `elasticsearch` — Elasticsearch, встраиваемая система для текстового поиска [29];
13. `hbase` — Apache HBase, распределённая табличная база данных [30];
14. `guava` — Google Guava, набор библиотек-расширений для Java [31];
15. `hadoop-common` — Apache Hadoop, экосистема для распределённого хранения и обработки данных (модуль ядра) [32];
16. `hadoop-hdfs` — тот же Hadoop (модуль распределённой файловой системы);
17. `hadoop-mapreduce` — тот же Hadoop (модуль для распределённых вычислений в парадигме MapReduce);

18. `hadoop-yarn` — тот же Hadoop (модуль планировщика ресурсов);

Параметры валидационных датасетов указаны в таблице 3.

Датасет	Количество формул	Средний размер формулы	Средняя глубина формулы	Доля выполнимых формул
`owasp`	30 935	3169.44	27.0	0.029
`cassandra`	13 896	164.84	11.73	0.057
`kafka`	46 867	5103.27	18.93	0.083
`spark-core`	14 504	5599.33	28.0	0.061
`spark-streaming`	69 254	1168.64	8.31	0.057
`utbot-core`	31 043	4570.6	28.95	0.061
`utbot-java`	12 492	116.4	10.71	0.045
`utbot-python`	24 893	266.66	6.04	0.115
`utbot-js`	13 276	2286.54	133.8	0.062
`utbot-go`	43 988	281.69	19.21	0.014
`zookeeper`	38 631	1196.51	47.23	0.046
`elasticsearch`	38 967	388.62	19.72	0.001
`hbase`	38 049	3513.78	90.81	0.019
`guava`	56 607	545.98	34.99	0.014
`hadoop-common`	41 255	419.05	10.32	0.081
`hadoop-hdfs`	109 294	2795.7	210.87	0.032
`hadoop-mapreduce`	12 044	1294.2	47.17	0.011
`hadoop-yarn`	32 555	120.71	13.01	0.035

Таблица 3: Параметры валидационных датасетов, собранных в процессе работы USVM.

Отмечу, что в валидационных датасетах также отобраны формулы, размер и глубина которых тоже не превышает 10 000 и 2 000 соответственно, но, помимо этого, здесь добавилось ещё одно условие: размер должен быть не меньше некоторого значения, которое подбиралось эмпирическим путём отдельно для каждого датасета (обычно оно находилось в районе 200–500, и его можно угадать, если посмотреть на средний столбец в таблице). Поэтому формулы из валидационных датасетов кажутся больше, чем из тренировочных. Это было сделано, чтобы уменьшить объём данных и ускорить вычисления, а также чтобы оценивать качество модели именно на больших формулах, так

как кажется, что на практике модель должна существенно помогать движку именно в этом случае.

Ещё внимательный читатель может заметить, что во всех датасетах из формул, собранных в процессе работы USVM, есть огромный дисбаланс классов: меньше десяти процентов формул являются выполнимыми. На самом деле, в этом нет ничего неожиданного, потому что в процессе символьного исполнения программы SMT-решателю действительно чаще всего приходится иметь дело с невыполнимыми формулами.

1.3. Метрики

Поскольку в работе рассматривается задача бинарной классификации, было принято решение следить за двумя метриками: ROC-AUC (площадь под ROC-кривой, далее будет обозначаться «ROC-AUC») и Average Precision (площадь под precision-recall кривой, далее будет обозначаться «AP») для случая, когда представителями положительного класса считаются выполнимые формулы. Пример на рис. 9.

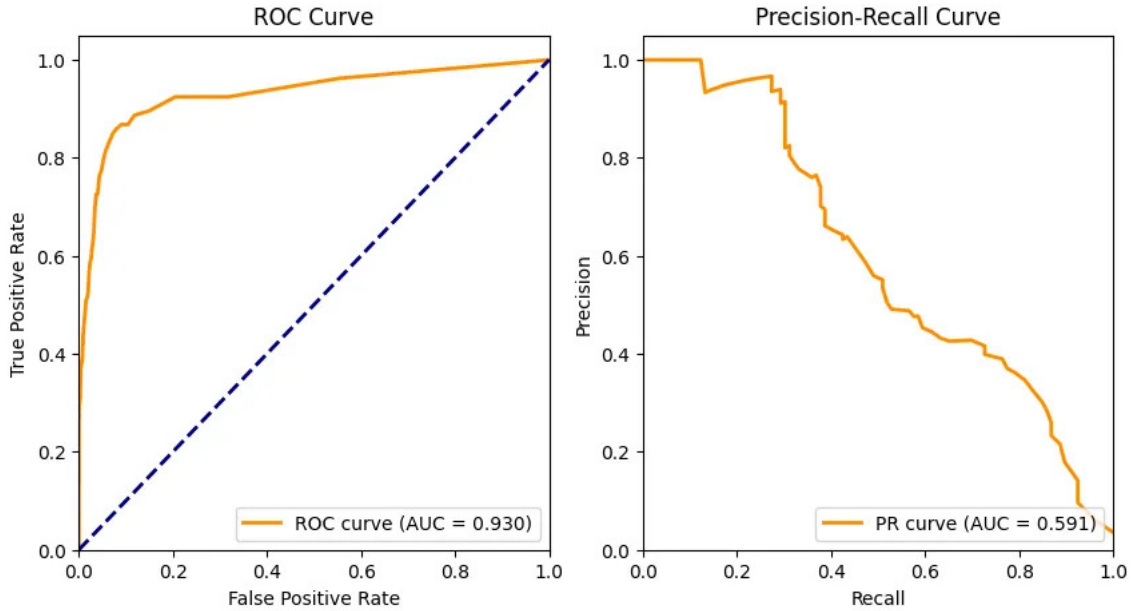

Рис. 9: Пример вычисления ROC-AUC и AP. Источник: `https://juandelacalle.medium.com/how-and-why-i-switched-from-the-roc-curve-to-the-precision-recall-curve-to-analyze-my-imbalanced-6171da91c6b8` (дата обр. 27.05.2024).

Использование ROC-AUC обусловлено тем, что, помимо классификационной, это ещё и ранжирующая метрика, а в работе будет также интересно смотреть и на ранжирующие способности модели. Использование AP обусловлено тем, что ROC-AUC начинает терять репрезентативность, если в выборке данных появляется большой дисбаланс классов [33].

Отмечу два факта, которые понадобятся в дальнейшем:

1. ROC-AUC для случайного предсказания, а также для предсказания всем примерам положительной или предсказания всем примерам отрицательной метки равен $0.5 \pm \varepsilon$.

2. AP для случайного предсказания, а также для предсказания всем примерам положительной или предсказания всем примерам отрицательной метки равен $t \pm \varepsilon$, где t — доля примеров положительного класса в выборке.

Исходя из этого, можно для каждого датасета посчитать базовые значения метрик и сравнивать полученные результаты с ними, чтобы показать, что решение имеет хоть какой-то смысл.

Таким образом, в представленных результатах будет указано по два значения ROC-AUC и AP — базовое, которое вычислено согласно только что изложенным фактам, и тестовое, которое получено при валидации обученной ранее модели на этом датасете. В таблицах с результатами базовое значение будет называться «контрольным», а полученное при валидации — «тестовым». Выбранные названия отсылают к описанию результатов A/B-тестов (однако, это только отсылки — в действительности, никакие A/B-тесты в данной работе не проводились).

Помимо этого, в одном из экспериментов будет считаться метрика «precision at fixed recall», которая выражает уровень точности при заданном уровне полноты. За этими значениями тоже полезно следить, так как для символьного движка выполнимые формулы (которые в нашей задаче являются представителями положительного класса) гораздо интереснее, чем невыполнимые.

2. Подход с использованием GNN

2.1. Архитектура нейронной сети для решения задачи

Придуманное в ходе работы решение объединяет в себе идеи из всех статей, рассмотренных в главе про обзор литературы и предметной области.

В качестве основной архитектуры для решения задачи была выбрана некоторая аппроксимация GNN, работающая схожим образом с тем, что было ранее описано в разделе GNN for Scheduling of SMT Solvers [19], и заимствующая некоторые идеи у NeuroSAT [20]. Здесь используется такая же схема передачи GNN-сообщений по данному на вход абстрактно-синтаксическому дереву (AST) формулы, превращённому в ориентированный ациклический граф для простоты вычислений, однако всё это происходит с некоторыми отличиями:

1. рёбра проводятся только в направлении от операндов к операторам (см. пример ниже);
2. обновление эмбеддингов состояний вершин начинается от истоков графа (листьев AST формулы) и производится в порядке топологической сортировки;
3. передача векторов-сообщений происходит согласно ориентации рёбер (от начала к концу);
4. вычисление состояния каждой вершины производится ровно один раз и только после вычисления состояний всех вершин, от которых она зависит, т. е. всех других вершин, из которых есть ребро в неё;
5. указанные вычисления для всех вершин делаются за один проход по формуле;
6. в конце в качестве итогового вектора-представления формулы используется вектор-состояние стока графа (корневой вершины формулы).

Для лучшего понимания, на рис. 10 изображён пример графа и того, в каком порядке производятся вычисления состояний вершин.

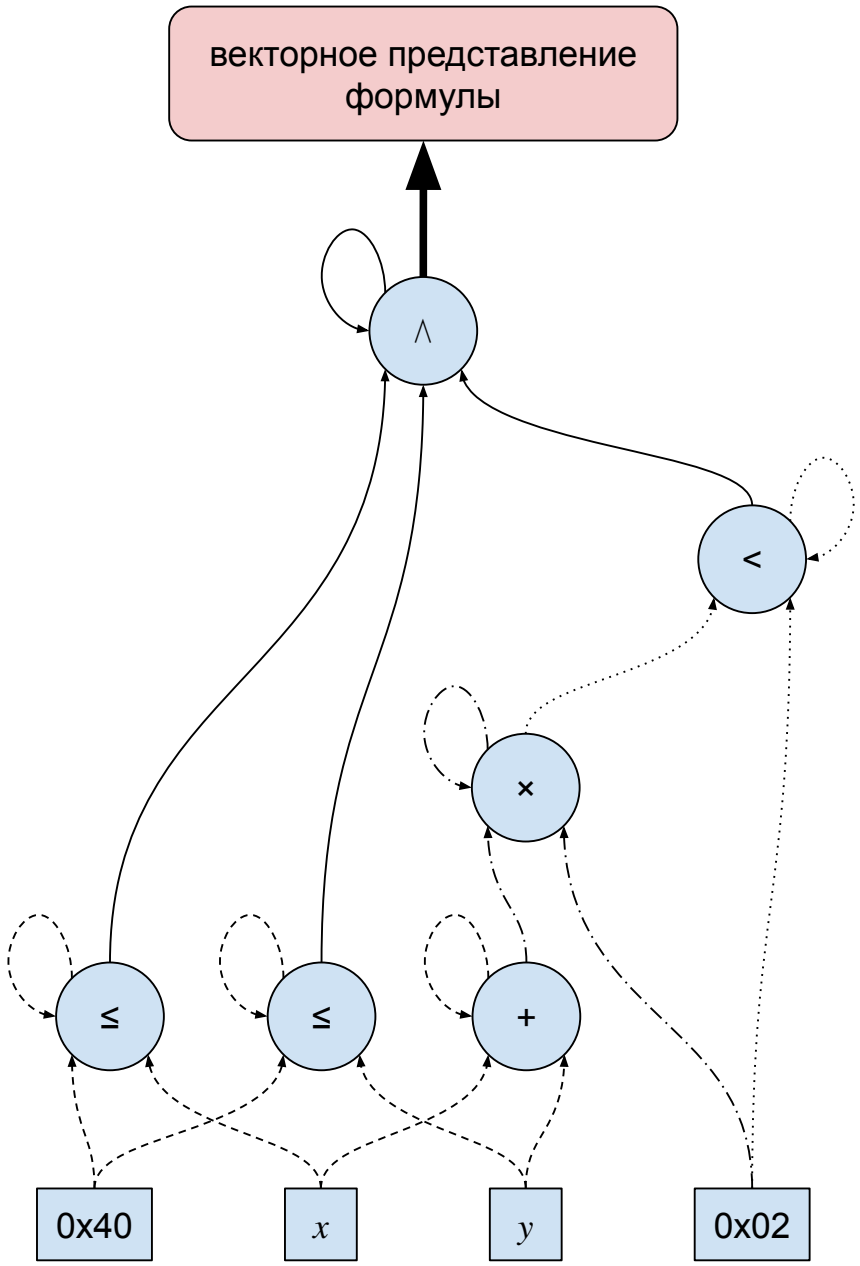

Рис. 10: Пример ориентированного ациклического графа, полученного из AST формулы $(\texttt{0x40} \leqslant x) \wedge (\texttt{0x40} \leqslant y) \wedge ((x + y) \times \texttt{0x02} < \texttt{0x02})$ в логике вычислений с восьмибитными векторами. Вершины разбиты по уровням согласно порядку вычислений состояний: снизу истоки (листья) — с них начинается вычисление, сверху сток (корень) — его вектор вычисляется последним, и он же считается итоговым вектором-представлением всей формулы. Начертание каждого ребра отражает момент, в который по нему производится передача сообщения: по рёбрам, нарисованным штриховой линией (- - -) передача производится на первом шаге, штрих-пунктирной линией (- · -) — на втором, пунктирной линией (· · ·) — на третьем, а сплошной (—) — на четвёртом. Петли в данном графе обозначают, что для вычисления состояния вершины также используется её начальное состояние (об этом подробнее написано в параграфах про начальные состояния вершин и вычисление их состояний).

Описанная выше структура была придумана мной при попытке оптимизировать методы вычислений из статьи [19], чтобы все нужные состояния можно было посчитать за один проход по AST формулы. Тем не менее, позже, в процессе очередной итерации исследования предметной области, выяснилось, что такая архитектура была придумана ещё в 1997 году, описана в статьях [34] и [35] и более известна как рекурсивная нейронная сеть или RvNN[22]. Авторы указанных статей пытались изобрести подход к обучению моделей для иерархически устроенных данных (рис. 11), чтобы можно было учитывать и их специфику наряду с тем, как это происходит в свёрточных сетях для данных в виде картинок или в рекуррентных сетях для данных в виде последовательностей. В итоге, было предложено использовать процедуру, похожую на передачу сообщений в GNN, применительно к дереву или ориентированному ациклическому графу, задающим саму иерархию в данных. В статье даже в качестве одного из примеров таких данных рассматриваются логические термы (рис. 12).

Однако с ростом доступных вычислительных мощностей данный подход уступил более обобщённому подходу с использованием GNN и не получил активного дальнейшего развития.

Посчитанный в ходе описанных выше действий вектор-представление графа считается эмбеддингом формулы, который потом можно передать в MLP для предсказания различных параметров формулы, например, её выполнимости. Именно так делается у меня, и ровно так же было предложено делать в статье [34] (рис. 13).

2.2. Начальные состояния вершин

Выше написано, что в вычислениях участвуют начальные векторы-состояния вершин, но до этого момента не было сказано, откуда они берутся.

С начальными векторами вершин-операций всё легко: эмбеддинг для каждой из них просто выучивается, как это делается, например, для слов в задаче обработки естественного языка, и сохраняются в специальную таблицу. Замечу, что на этом этапе можно сделать разделение одной операции,

[22] От англ. *Recursive Neural Network*.

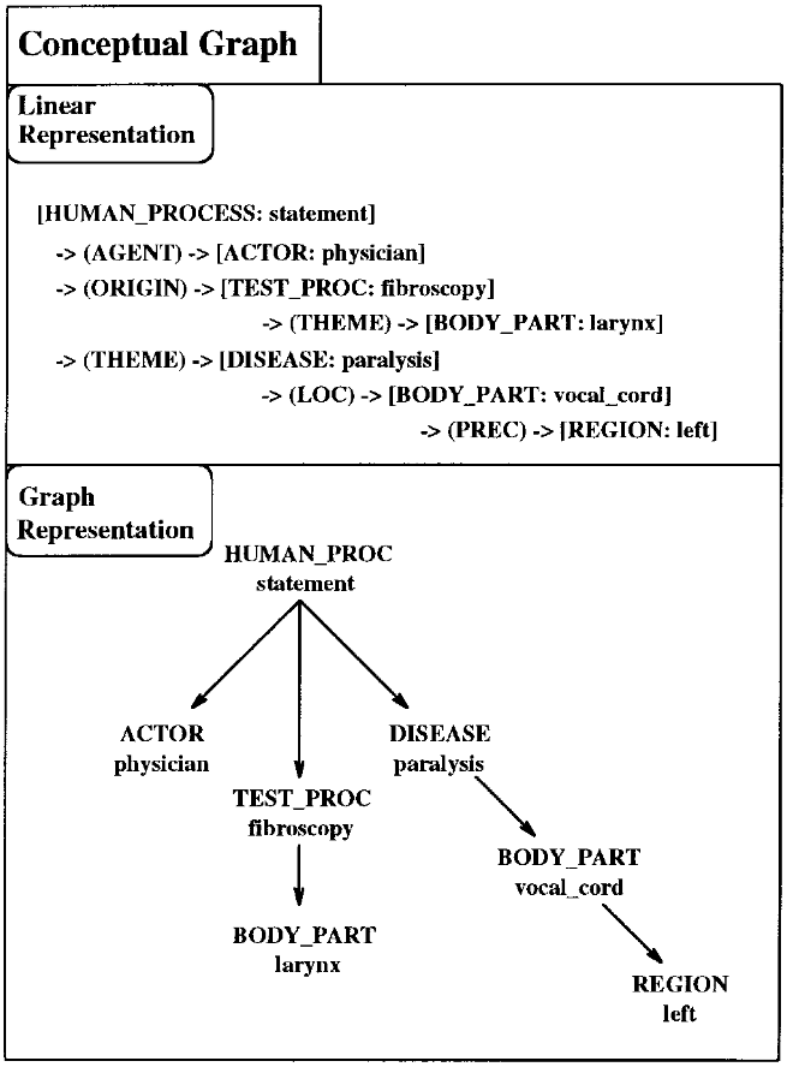

Рис. 11: Пример иерархически устроенных данных: анамнез пациента. Картинка взята из статьи [34].

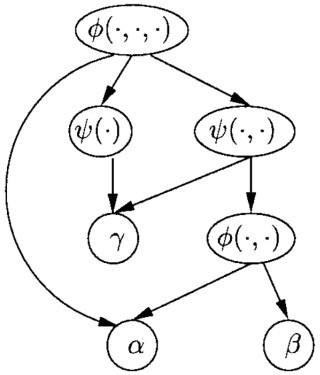

Рис. 12: Пример представления логического терма $\phi(\alpha, \psi(\gamma), \psi(\gamma, \phi(\alpha, \beta)))$ в виде ориентированного ациклического графа. Картинка взята из статьи [35].

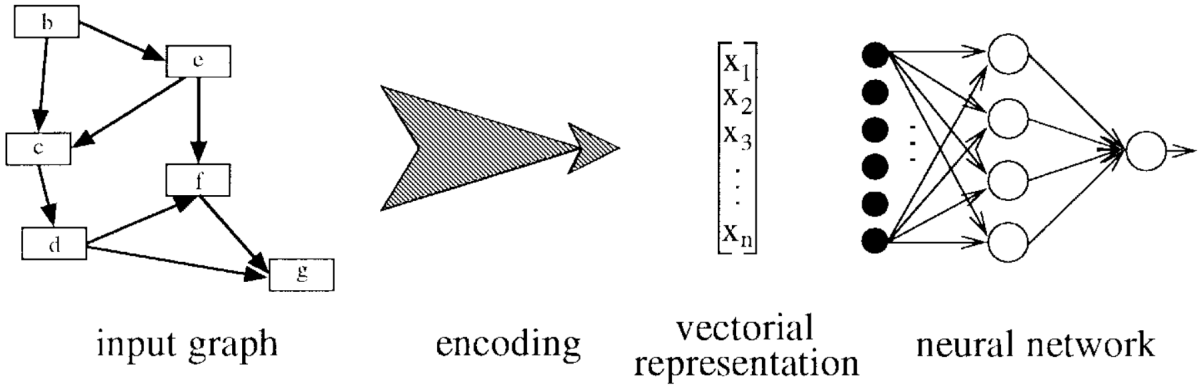

Рис. 13: Общий подход к решению задачи предсказания произвольных параметров ориентированного ациклического графа. Картинка взята из статьи [34].

работающей с разными типами данных, на разные, чтобы учесть отличия в механизме их работы. Например, сложение целых чисел, сложение вещественных чисел и сложение битовых векторов можно считать разными операциями.

С начальными векторами вершин-переменных и вершин-констант всё куда сложнее. Начальный вектор для вершины-переменной должен содержать некоторую информацию о том, что эта вершина соответствует переменной определённого типа, однако с этим есть несколько проблем. Во-первых, векторы, кодирующие разные переменные должны отличаться друг от друга, чтобы как-то отразить то, что в выполняющем наборе формулы им могут быть присвоены разные значения. Во-вторых, эти векторы нельзя основывать на информации об именах переменных, поскольку обычно имя переменной не несёт в себе никакого смысла и используется для разделения переменных внутри одной формулы, да и сами логические формулы инвариантны относительно переименования переменных.

В процессе работы было придумано три возможных варианта действий с переменными:

1. Можно проигнорировать все имена переменных в формуле, оставив только информацию об их типе. После такого преобразования все переменные можно считать конструкциями вида `Var[Тип]`, поэтому в формуле вместо имён переменных будут записи `Var[Integer]`, `Var[Real]`, `Var[BitVec<8>]`, `Var[Array<BitVec<12>, Float64>]` и прочие подобные. Таких записей будет немного, поэтому для них можно выучить эм-

беддинги подобно тому, как это делается для вершин-операций. Такой подход не учитывает различия между переменными, однако позволяет представить формулу в хоть каком-нибудь виде. Например, в формуле на рис. 10 у переменных x и y будет одинаковое начальное состояние, соответствующее эмбеддингу, выученному для записи `Var[BitVec<8>]`.

2. Можно завести отдельную табличку с набором эмбеддингов для переменных и на каждом запуске модели при обучении или при использовании выдавать каждой переменной случайный эмбеддинг из этого набора. Такая схема будет предоставлять разные векторы разным переменным и не будет привязывать их к именам этих переменных, чтобы учесть свойство инвариантности формул относительно переименования. Предполагается, что процессе обучения векторы будут сходиться к случайным точкам, которые достаточно хорошо описывают семантику переменных, а модель научится определённым образом подстраиваться под эту случайность. Оно может сработать, поскольку разнообразные вероятностные техники периодически используются в машинном обучении (например, существует техника Negative Sampling [36]). Это даже можно считать способом регуляризацией модели.

3. Помимо этого, можно сделать так же, как в предыдущем пункте, но не выучивать табличку с эмбеддингами, а вместо этого случайным образом нумеровать переменные натуральными числами, после чего для этих номеров применять технику позиционного кодирования (англ. *positional encoding*) из знаменитой статьи «Attention Is All You Need» [37]. Идея работает такая же, как и пункт 2, но имеет меньше обучаемых параметров и потенциально лучше обобщается на формулы большого размера.

Если про технику работы с переменными всё более-менее понятно (хотя бы понятно, в каком направлении развивать решение), то константы в формуле представляют настоящую проблему, так как нужно научиться переводить и целые числа, и вещественные числа, и битовые векторы, и массивы значений в некоторое многомерное пространство с сохранением всей семантики: абсолютных значений, отношений больше-меньше, внутренней структуры (для

битовых векторов и массивов). Задача построения такого представления оказалась слишком сложной, и за время работы не было придумано никаких методов, которые, в теории, могли бы дать существенный прирост качества. Тем не менее, здесь также можно выписать три варианта, что можно сделать с кодированием констант:

1. Можно поступить аналогично первому варианту подхода к переменным — убрать значения всех констант и оставить вместо них только конструкции вида `Val[Тип]`, после чего обучать эмбеддинги для этих конструкций. Например, в формуле на рис. 10 у значений `0x02` и `0x40` будет одинаковое начальное состояние, соответствующее эмбеддингу, выученному для записи `Val[BitVec<8>]`.

2. Можно воспользоваться способом, предложенным в статье [38] — разбивать пространство значений, из которого приходит константа (например, для вещественной переменной это будет вещественная прямая), на группы (бины) и строить вектор, как это показано на рис. 14. Такой способ, к сожалению, не применим для битовых векторов, так как они могут иметь произвольный размер, а без априорного знания про этот размер отобразить вектор в линейное пространство с сохранением семантики всех операций невозможно.

3. Помимо этого, можно воспользоваться техникой под названием Fourier Feature Mapping [39] [40] — вторым способом, предложенным в статье [38]. Эта статья исследует способы кодирования семантики произвольных числовых значений и ставит своей целью приблизить нейронные сети к градиентным бустингам по качеству решения задач табличного машинного обучения. Насколько я понял, техника заключается в вычислении некоторого количества дискретных преобразований Фурье с разными частотами и использовании полученных значений в качестве координат вектора-эмбеддинга.

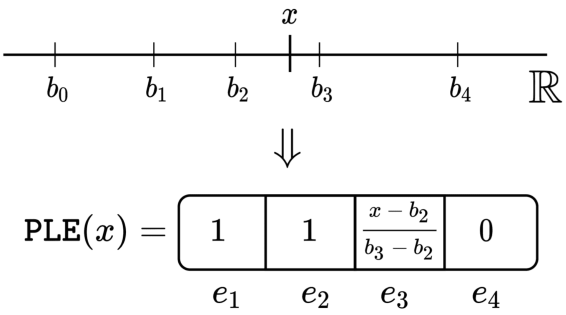

Рис. 14: Построение эмбеддинга вещественной константы с помощью бинов. Картинка взята из статьи [38].

2.3. Вычисление состояний вершин

Теперь пришло время поговорить про то, каким способом, собственно, вычислять вектор-состояние очередной вершины. Согласно схеме обновления состояний в GNN это должно происходить посредством передачи векторов-сообщений от соседних вершин. Осталось только выбрать, какие функции подставить в формулу (5). В экспериментах участвовали два варианта.

2.3.1 SAGE Convolution

Первый был представлен в статье [41] и называется GraphSAGE. В нём вычисление происходит согласно формуле (6), где x_v — изначальное состояние вершины v, а x'_v — новое, которое вычисляется в процессе.

$$x'_v = \sigma\left(W_3 \cdot \left(W_1 \cdot x_v + W_2 \cdot \operatorname*{MEAN}_{u \in \mathcal{N}(v)} x_u\right) + b\right) \tag{6}$$

Здесь $\mathcal{N}(v)$ — окрестность вершины v, функция MEAN обозначает взятие среднего значения по указанному множеству, σ — сигмоидная функция[23], а матрицы W_1, W_2, W_3 и вектор b являются выучиваемыми параметрами сети.

[23] $\sigma(x) = \dfrac{1}{1 + e^{-x}}$

2.3.2 Transformer Convolution

Второй подход был представлен в статье [42] и называется Transformer Graph Convolution. В нём вычисление происходит согласно формулам (7) и (8).

$$\alpha_u = \underset{u \in \mathcal{N}(v)}{\text{SOFTMAX}} \left[\frac{(W_3 \cdot x_v)^T (W_4 \cdot x_u)}{\sqrt{d}} \right] \tag{7}$$

$$x'_v = W_1 \cdot x_v + \sum_{u \in \mathcal{N}(v)} \alpha_u W_2 \cdot x_u \tag{8}$$

Здесь d — размерность участвующих в вычислениях векторов, функция SOFTMAX по указанному множеству обозначает softmax-преобразование[24] чисел этого множества, а матрицы W_1, W_2, W_3 и W_4 являются выучиваемыми параметрами сети.

Нетрудно заметить, что механизм вычислений похож на тот, что используется в архитектуре Transformer [37] (собственно, оттуда и название подхода): значения α являются коэффициентами механизма внимания, матрица W_3 формирует вектор-запрос, W_4 — вектор-ключ, а W_2 — вектор-значение.

2.3.3 Учёт начального состояния вершины

Из формул (6) и (8) видно, что для вычисления вектора-состояния вершины используется её же начальное состояние (в указанных формулах оно домножается на матрицу W_1). Именно таким образом и происходит учёт информации об операции, которой соответствует вершина: изначально в вершину кладётся вектор, представляющий семантику этой операции, а потом он используется при вычислении состояния этой вершины через состояния её детей. Петли в графе на рис. 10 обозначают в точности это.

[24] $\text{SOFTMAX}(a_1, a_2, \ldots, a_n)_k = \dfrac{e^{a_k}}{\sum\limits_{i=1}^{n} e^{a_i}}$

2.4. Классификатор и функция потерь

С точки зрения нейронной сети задача поставлена как задача классификации, поэтому после прохода по графу и вычисления всех состояний вектор-состояние корневой вершины отправляется в классификатор, который представляет из себя нейронную сеть с четырьмя полносвязными слоями и слоями активации ReLU между ними.

В качестве функции потерь используется бинарная кросс-энтропия.

3. Эксперименты и результаты

К сожалению, из-за присутствовавших трудностей с вычислительными мощностями все эксперименты проводились с использованием векторов-состояний вершин размера исключительно 32 (другие размеры попробовать не удалось), а из описанных в разделе 2.2 методов построения начальных состояний вершин были попробованы только первые методы построения состояний для переменных и констант. Однако, даже с такими подходами удалось провести ряд экспериментов и получить некоторые практически значимые результаты.

3.1. Первый эксперимент

В первом эксперименте была предпринята попытка обучить модель с архитектурой SAGE Convolution, описанной в разделе 2.3.1, на датасетах из формул с SMT-COMP (таблица 1).

Для полноты картины эксперимент был устроен следующим образом: было обучено три модели (по одной на каждом из датасетов), после чего для каждой полученной модели считались метрики на её обучающем датасете, на его частных случаях (например, `BitVec` $\subset$ `SymbEx`) или на его альтернативных вариантах (например, для `SymbEx` был взят датасет `usvm-test`, из формул, собранных с символьной машины [11]). Это сделано, чтобы, в том числе, узнать, может ли знание о более сложных формулах помочь модели разбираться с более простыми.

Разумеется, в случае, когда модель обучалась на датасете `X`, подсчёт метрик на этом датасете производится на специально отложенной тестовой выборке[25]. Если датасет `X` не участвовал в обучении модели, то метрики считались на всех его данных. Это соблюдается в этом и во всех последующих экспериментах.

Обучение каждой модели производилось в течение 50 эпох с помощью оптимизатора Adam [43] с learning rate[26], изначально равным 10^{-4}, и управ-

[25] В этом и во всех последующих экспериментах в каждом датасете примерно 15% формул были отложены в валидационную выборку, и ещё примерно 10% — в тестовую.

[26] Так называют длину шага градиентного спуска.

ляемым согласно динамическому расписанию *Reduce LR On Plateau*.

В течение всего обучения проводился отбор лучших моделей (англ. *model selection*) по значению функции потерь на валидационной выборке.

Результаты отображены в таблице 4. К сожалению, они получились не особо впечатляющими — во время обучения даже на тренировочном датасете функция потерь упиралась в границу, которую никак не могла преодолеть, так что модель выглядит недообученной. Оно и понятно — модель не умеет отличать переменные друг от друга и не понимает семантику констант в формуле.

Трен. датасет	Вал. датасет	Контр-ный ROC-AUC	Тестовый ROC-AUC	Контр-ный AP	Тестовый AP
`BitVec`	`BitVec`	0.500	0.718	0.378	0.695
`SymbEx`	`BitVec`	0.500	0.548	0.378	0.426
	`SymbEx`	0.500	0.358	0.610	0.678
	`usvm-test`	0.500	0.503	0.038	0.054
`QuaFree`	`BitVec`	0.500	0.719	0.378	0.706
	`SymbEx`	0.500	0.592	0.610	0.716
	`QuaFree`	0.500	0.564	0.622	0.705

Таблица 4: Метрики, полученные при обучении SAGE Convolution на датасетах с SMT-COMP [21].

3.2. Второй эксперимент

Второй эксперимент заключался в обучении модели с той же архитектурой, что и в первом эксперименте, но уже на других датасетах. В этот раз были рассмотрены датасеты из формул, собранных с символьного движка USVM [11] (таблица 2).

Аналогично первому эксперименту, здесь модель обучалась на каком-то датасете, после чего валидировалась, в том числе, на некоторых других датасетах, однако на этот раз валидационные датасеты, в основном, не пересекались с тренировочными. Сделано это было, чтобы можно было оценить обобщающую способность модели. Подробнее об этом было написано во втором абзаце раздела 1.2.

Обучение каждой модели производилось в течение 30 эпох с помощью оптимизатора AdamW [44] с параметром weight decay[27], равным 10^{-3}, и learning rate, изначально равным 10^{-4}, и управляемым согласно динамическому расписанию *Reduce LR On Plateau*. Такое большое значение weight decay было продиктовано желанием как можно сильнее повысить обобщающую способность модели, чтобы она потенциально могла показывать хорошее качество на данных из другого распределения.

В течение всего обучения проводился отбор лучших моделей по значению функции потерь на валидационной выборке.

Результаты в таблице 5.

Трен. датасет	Вал. датасет	К. ROC-AUC	Т. ROC-AUC	К. AP	Т. AP
`usvm-test`	`usvm-test`	0.500	0.873	0.038	0.566
	`the-algorithms`	0.500	0.551	0.066	0.067
	`usvm-core`	0.500	0.641	0.066	0.087
	`owasp-all`	0.500	0.314	0.029	0.101
`usvm-test` & `the-algorithms`	`usvm-test` & `the-algorithms`	0.500	0.786	0.053	0.447
	`usvm-core`	0.500	0.833	0.066	0.198
	`owasp-all`	0.500	0.636	0.029	0.260
`usvm-test` & `the-algorithms` & `usvm-core`	`usvm-test` & `the-algorithms` & `usvm-core`	0.500	0.821	0.058	0.593
	`owasp-all`	0.500	0.958	0.029	0.873

Таблица 5: Метрики, полученные при обучении SAGE Convolution на датасетах, собранных с USVM [11]. Символ «&» здесь обозначает объединение датасетов. «К.» обозначает контрольное значение, «Т.» — тестовое.

Здесь метрики выглядят куда интереснее, чем в первом эксперименте. Во-первых, полученные числа сами по себе выглядят более солидно. Во-вторых, хорошо видно, как растёт обобщающая способность модели, если обучать её на более сложных и разнообразных формулах. Особенно, это за-

[27]Параметр отвечает за регуляризацию модели на основе штрафа за слишком большие абсолютные значения весов.

метно по метрикам на датасете `owasp-all` (версия датасета `owasp`, из которой не убирали формулы слишком маленького размера).

3.3. Третий эксперимент

Удачные результаты второго эксперимента (особенно полученные на датасете `owasp-all`) натолкнули на мысль, что, возможно, последняя обученная модель (нижняя в таблице 5) уже достаточно хороша, чтобы уметь предсказывать ответ для формул, возникающих в процессе анализа *произвольных* программ с помощью символьного движка USVM.

Для проверки этой гипотезы были собраны датасеты из формул, возникших в процессе анализа разных проектов с открытым исходным кодом, написанных на JVM-языках (таблица 3), и на них были посчитаны уже знакомые нам метрики. Результаты в таблице 6.

Думаю, что результаты достаточно хороши, для того, чтобы их можно было считать в значительной степени положительными, и чтобы полученную модель можно было использовать на практике.

Поскольку обычно в процессе символьного исполнения распознавание выполнимых формул важнее, чем распознавание невыполнимых, я также привожу результаты метрик «precision @ fixed recall» (уровень точности при заданном уровне полноты) в таблице 7 в приложении.

Тем не менее, видно, что результаты на датасетах `BitVec` и `SymbEx` с SMT-COMP (таблица 1), в которых содержатся формулы из аналогичных логик, оставляют желать лучшего. Так что, скорее всего, высокие результаты на оставшихся датасетах обусловлены, в первую очередь, тем, что в процессе работы движка USVM получаются сильно специфические формулы, по структуре которых легко предсказать их выполнимость. Однако, это не повод не использовать данную особенность при решении практической задачи.

3.4. Четвёртый эксперимент

Для полноты исследования было также решено попробовать обучить модель с другой архитектурой (Transformer Convolution из раздела 2.3.2), однако этот эксперимент оказался неудачным.

Датасет	Контрольный ROC-AUC	Тестовый ROC-AUC	Контрольный AP	Тестовый AP
BitVec	0.500	0.613	0.378	0.484
SymbEx	0.500	0.509	0.610	0.628
owasp	0.500	0.914	0.029	0.245
cassandra	0.500	0.956	0.057	0.603
kafka	0.500	0.827	0.083	0.689
spark-core	0.500	0.970	0.061	0.616
spark-streaming	0.500	0.992	0.057	0.889
utbot-core	0.500	0.982	0.061	0.657
utbot-java	0.500	0.997	0.045	0.911
utbot-python	0.500	0.923	0.115	0.480
utbot-js	0.500	0.801	0.062	0.307
utbot-go	0.500	0.996	0.014	0.712
zookeeper	0.500	0.840	0.046	0.407
elasticsearch	0.500	0.999	0.001	0.468
hbase	0.500	0.824	0.019	0.083
guava	0.500	0.998	0.014	0.816
hadoop-common	0.500	0.975	0.081	0.714
hadoop-hdfs	0.500	0.847	0.032	0.379
hadoop-mapreduce	0.500	0.948	0.011	0.198
hadoop-yarn	0.500	0.994	0.035	0.845

Таблица 6: Результаты валидации на датасетах из реальных формул, собранных с символьной машины USVM [11] (см. таб. 3).

При обучении на датасетах с SMT-COMP (таблица 1) где-то после седьмой-восьмой эпохи модель начинала бесконтрольно переобучаться (метрики на тренировочном наборе уверенно улучшались, а на валидационном стремительно ухудшались). Причём лучшая модель, полученная на тот момент, всё равно проигрывала модели с архитектурой SAGE Convolution.

Более того, в архитектуру Transformer Convolution встроена возможность использовать механизм dropout [45] при обучении, и подобная проблема проявлялась даже при вероятности отключения нейрона $p = 0.2,\ 0.3,\ 0.4,\ 0.5$. Использование техники Layer Normalization [46], которая часто помогает при проблемах с обучением механизма внимания, тоже не дало улучшений.

Видимо, действительно, данные в датасетах с SMT-COMP устроены

настолько сложно и отражают настолько общий случай, что никакие техники, успешно показывающие себя на датасетах с USVM (раздел 1.2), здесь не работают.

Сами датасеты, собранные с USVM, в этом эксперименте не использовались, и качество на них не измерялось, поскольку модель с такой архитектурой будет иметь значительные проблемы с производительностью, поэтому получение прироста качества за счёт перехода на такую архитектуру на практике может обернуться замедлением работы всей системы и, как следствие, оказаться бесполезным.

4. Дальнейшее развитие

Несмотря на неплохие результаты на некоторых имеющихся датасетах, модель можно улучшить во многих местах. Вот список направлений, по которым стоит развивать проект:

1. Попробовать провести эксперименты с разными гиперпараметрами нейронной сети: попробовать другие (отличные от 32) значения размерности вектора-состояния вершины, другое количество линейных слоёв в классификаторе и т. д..

2. Разумеется, стоит попробовать использовать более содержательные из описанных в разделе 2.2 техник построения векторного представления переменных и констант в формуле.

3. Можно запускать передачу сообщений по графу формулы не только в одном направлении, а сразу во всех, как это было сделано в статье [19], а после этого использовать не только состояние корневой вершины, а агрегировать информацию сразу со всех вершин.

4. Чтобы увеличить датасет и повысить обобщающую способность модели, можно попробовать применить разного рода аугментации к формулам в тренировочном датасете.

5. Если получится научиться проверять на выполнимость относительно произвольные формулы, можно озаботиться поддержкой формул с кванторами.

6. Поскольку использование модели на практике предполагает не столько предсказательную способность модели, сколько ранжирующую, можно попробовать применить техники из этой области. Например: учитывать косинусное расстояние между вычисляемыми векторами, представляющими выполнимые и невыполнимые формулы, использовать ранжирующую функцию потерь LambdaRank [47] или функцию потерь Triplet Loss для задач информационного поиска [48] [49].

7. Нынешняя версия модели не использует имена переменных, считая, что они ничего толкового не значат. Тем не менее, если речь идёт о модели, используемой для распознавания формул, которые возникают при работе USVM, можно попробовать учитывать имена некоторых переменных, так как они порождаются определённым образом, и в них может содержаться какая-то полезная информация.

8. Если говорить об использовании модели на практике, стоит отметить необходимость её эффективной реализации. Текущая версия модели написана на языках Python и Kotlin и значительно уступает современным SMT-решателям, написанным на языке C++. Помимо этого можно применить различные техники оптимизации вычислений в нейронных сетях как, например, квантизация.

9. Наконец, после получения быстрой, показывающей хорошее качество на практике модели, можно озаботиться встраиванием её в работу символьного движка USVM [11]. Для этого нужно будет придумать ряд эвристик для выбора состояния, которые будут учитывать предсказание модели. Изначально такая задача не предполагалась, но она является естественным и логичным продолжением исследования, проведённого в этой работе.

Заключение

В рамках данной работы была предпринята попытка создать нейронную сеть для предсказания выполнимости SMT-формулы: собрать подходящий датасет, реализовать архитектуру нейронной сети, провести ряд экспериментов с обучением и валидацией моделей и сделать выводы.

Я считаю, что сформулированная в работе задача выполнена — основные шаги для решения поставленной поставленной задачи, описанные в соответствующем разделе, пройдены, и требуемое исследование проведено.

В процессе работы было выяснено, что для решения задачи в общем случае протестированных методов недостаточно, и нужно использовать более умные подходы к построению векторного представления формулы, о чём написано в главе 4.

Однако, в контексте практического применения (решения формул, возникающих в процессе символьного исполнения на движке USVM) данная задача может быть решена с относительно хорошими результатами при помощи более-менее базовых методов (таблицы 5, 6 и 7). Скорее всего, это происходит из-за того, что в практической задаче возникают очень специфические формулы, которые значительно отличаются от общего случая в лице SMT-COMP, и для которых гораздо проще предсказывать ответ из-за того, что модель привязывается на некоторые особенности возникающих формул. Я думаю, что перспектива практического использования полученных результатов во многом может опираться на этот факт.

Также в процессе исследования было протестировано несколько возможных архитектур нейронной сети (разделы 2.3.1 и 2.3.2).

Сравнение полученного решения с аналогами не проводилось, поскольку таковых на данный момент не существует (я не считаю аналогом модель NeuroSAT [20], так как в той задаче существенно отличаются ограничения и ожидаемый результат).

Проведённое исследование можно существенно расширять и дополнять. О методах и возможных путях развития написано в главе 4.

Список литературы

[1] SMT-LIB 2 Logics. URL: https://smt-lib.org/logics.shtml (дата обр. 13.05.2024).

[2] Leonardo de Moura and Nikolaj Bjørner. «Z3: an efficient SMT solver». In proceedings of *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS '2008), pp. 337–340. Springer, 2008.

[3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli and Yoni Zohar; Dana Fisman (ed.) and Grigore Rosu (ed.). «cvc5: A Versatile and Industrial-Strength SMT Solver». In proceedings of *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS '2022), part I, pp. 415–442. Springer, 2022.

[4] Bruno Dutertre; Armin Biere (ed.) and Roderick Bloem (ed.). «Yices 2.2». In proceedings of *Computer-Aided Verification* (CAV '2014), pp. 737–744. Springer, 2014.

[5] Aina Niemetz and Mathias Preiner; Constantin Enea (ed.) and Akash Lal (ed.). «Bitwuzla». In proceedings of *Computer-Aided Verification* (CAV '2023), part II, pp. 3–17. Springer, 2023.

[6] Clark Barrett, Leonardo de Moura and Aaron Stump. «SMT-COMP: Satisfiability modulo theories competition». In proceedings of *Computer-Aided Verification* (CAV '2005), pp. 503–516. Springer, 2005.

[7] SMT-COMP. URL: https://smt-comp.github.io/ (дата обр. 15.05.2024).

[8] James C. King. «Symbolic execution and program testing». In *Communications of the ACM*, July 1976, volume 19, number 7, pp. 385–394. Association for Computing Machinery, 1976.

[9] KLEE Symbolic Execution Engine. URL: https://klee-se.org/ (дата обр. 15.05.2024).

[10] Cristian Cadar, Daniel Dunbar and Dawson Engler. «KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs». In proceedings of *the 8th USENIX conference on Operating systems design and implementation* (OSDI '2008), pp. 209–224. USENIX Association, 2008.

[11] Поспелов Сергей Андреевич. «Проектирование и разработка универсальной символьной виртуальной машины». *Бакалаврская выпускная квалификационная работа*. URL: `http://hdl.handle.net/11701/42155`. СПбГУ, 2023.

[12] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner and Gabriele Monfardini. «The Graph Neural Network Model». In *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, Jan. 2009. IEEE, 2009.

[13] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals and George E. Dahl. «Neural Message Passing for Quantum Chemistry». *ArXiv abs:1704.01212* (2017).

[14] Michael M. Bronstein, Joan Bruna, Taco Cohen and Petar Veličković. «Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges». *ArXiv abs:2104.13478* (2021).

[15] Yujia Li, Daniel Tarlow, Marc Brockschmidt and Richard Zemel. «Gated Graph Sequence Neural Networks». *ArXiv abs:1511.05493* (2016).

[16] Thomas N. Kipf and Max Welling. «Semi-Supervised Classification with Graph Convolutional Networks». *ArXiv abs:1609.02907* (2017).

[17] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li and Maosong Sun. «Graph Neural Networks: A Review of Methods and Applications». *ArXiv abs:1812.08434* (2021).

[18] Mislav Balunovic, Pavol Bielik and Martin T. Vechev. «Learning to Solve SMT Formulas». *Advances in Neural Information Processing Systems 31* (2018).

[19] Jan Hůla, David Mojžíšek and Mikoláš Janota. «Graph Neural Networks for Scheduling of SMT Solvers». *IEEE 33rd International Conference on Tools with Artificial Intelligence* (ICTAI '2021), pp. 447–451. IEEE, 2021.

[20] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo Mendonça de Moura and David L. Dill. «Learning a SAT Solver from Single-Bit Supervision». *ArXiv abs:1802.03685* (2018).

[21] SMT-COMP 2023 benchmarks. URL: `https://smt-comp.github.io/2023/benchmarks.html` (дата обр. 20.05.2024).

[22] The Algorithms — Java. URL: `https://github.com/TheAlgorithms/Java` (дата обр. 20.05.2024).

[23] OWASP benchmark. URL: `https://owasp.org/www-project-benchmark/` (дата обр. 21.05.2024).

[24] Apache Cassandra. URL: `https://cassandra.apache.org/_/index.html` (дата обр. 21.05.2024).

[25] Apache Kafka. URL: `https://kafka.apache.org/` (дата обр. 21.05.2024).

[26] Apache Spark. URL: `https://spark.apache.org/` (дата обр. 21.05.2024).

[27] UnitTestBot. URL: `https://github.com/UnitTestBot/UTBotJava` (дата обр. 21.05.2024).

[28] Apache Zookeeper. URL: `https://zookeeper.apache.org/` (дата обр. 21.05.2024).

[29] Elasticsearch. URL: `https://www.elastic.co/elasticsearch` (дата обр. 21.05.2024).

[30] Apache HBase. URL: `https://hbase.apache.org/` (дата обр. 21.05.2024).

[31] Google Guava. URL: https://github.com/google/guava (дата обр. 21.05.2024).

[32] Apache Hadoop. URL: https://hadoop.apache.org/ (дата обр. 21.05.2024).

[33] Takaya Saito and Marc Rehmsmeier. «The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets». In *PLoS ONE*, vol. 10, 2015.

[34] Alessandro Sperduti and Antonina Starita. «Supervised neural networks for the classification of structures». In *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 714–735, May 1997. IEEE, 1997.

[35] Paolo Frasconi, Marco Gori and Alessandro Sperduti. «A general framework for adaptive processing of data structures». In *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 768–786, Sept. 1998. IEEE, 1998.

[36] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado and Jeffrey Dean. «Distributed Representations of Words and Phrases and their Compositionality». *ArXiv abs:1310.4546* (2013).

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin. «Attention Is All You Need». *ArXiv abs:1706.03762* (2017).

[38] Yury Gorishniy, Ivan Rubachev and Artem Babenko. «On Embeddings for Numerical Features in Tabular Deep Learning». *Neural Information Processing Systems* (NeurIPS ’2022).

[39] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron and Ren Ng. «Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains». *ArXiv abs:2006.10739* (2020).

[40] Yang Li, Si Si, Gang Li, Cho-Jui Hsieh and Samy Bengio. «Learnable Fourier Features for Multi-Dimensional Spatial Positional Encoding». *ArXiv abs:2106.02795* (2021).

[41] William L. Hamilton, Rex Ying and Jure Leskovec. «Inductive Representation Learning on Large Graphs». *ArXiv abs:1706.02216* (2017).

[42] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang and Yu Sun. «Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification». *ArXiv abs:2009.03509* (2021).

[43] Diederik P. Kingma and Jimmy Ba. «Adam: A Method for Stochastic Optimization». *ArXiv abs:1412.6980* (2014).

[44] Ilya Loshchilov and Frank Hutter. «Decoupled Weight Decay Regularization». *ArXiv abs:1711.05101* (2017).

[45] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov. «Dropout: a simple way to prevent neural networks from overfitting». In *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, June 2014. JMLR, 2014.

[46] Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey E. Hinton. «Layer Normalization». *ArXiv abs:1607.06450* (2016).

[47] Christopher Burges. «From ranknet to lambdarank to lambdamart: An overview». (2010).

[48] Matthew Schultz and Thorsten Joachims. «Learning a Distance Metric from Relative Comparisons». *Neural Information Processing Systems* (NeurIPS '2003).

[49] Gal Chechik, Varun Sharma, Uri Shalit and Samy Bengio. «Large Scale Online Learning of Image Similarity Through Ranking». In *Journal of Machine Learning Research*, vol. 11, pp. 11–14, June 2009. JMLR, 2010.

Приложение

Датасет	prc@.75	prc@.90	prc@.95	prc@.99	prc@1.
BitVec	0.456	0.417	0.402	0.385	0.378
SymbEx	0.619	0.619	0.619	0.612	0.610
owasp	0.146	0.124	0.124	0.115	0.110
cassandra	0.380	0.372	0.369	0.357	0.057
kafka	0.138	0.138	0.138	0.138	0.137
spark-core	0.490	0.490	0.490	0.485	0.485
spark-streaming	0.791	0.766	0.737	0.416	0.383
utbot-core	0.659	0.659	0.659	0.639	0.522
utbot-java	0.870	0.870	0.850	0.850	0.850
utbot-python	0.482	0.476	0.463	0.441	0.430
utbot-js	0.149	0.100	0.082	0.075	0.073
utbot-go	0.664	0.577	0.464	0.356	0.356
zookeeper	0.123	0.079	0.068	0.058	0.053
elasticsearch	0.156	0.131	0.105	0.080	0.080
hbase	0.061	0.047	0.041	0.022	0.021
guava	0.805	0.801	0.659	0.421	0.387
hadoop-common	0.637	0.631	0.631	0.369	0.300
hadoop-hdfs	0.173	0.072	0.033	0.033	0.032
hadoop-mapreduce	0.101	0.080	0.044	0.037	0.037
hadoop-yarn	0.789	0.668	0.629	0.541	0.352

Таблица 7: Результаты метрик «precision @ fixed recall» (уровень точности при заданном уровне полноты) на валидационных датасетах, собранных с символьной машины USVM (см. таб. 3).