

Санкт-Петербургский государственный университет

Чистякова Анна Артуровна

Выпускная квалификационная работа

Создание графовой нейронной сети,
предсказывающей пути при символьном
исполнении



Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование
информационных систем»*

Основная образовательная программа *СВ.5162.2021 «Технологии программирования»*

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., С. В. Григорьев

Рецензент:
ведущий разработчик ООО «ИИнтелКод» М. П. Костицын

Санкт-Петербург

2025

Saint Petersburg State University

Anna Chistyakova

Bachelor's Thesis

Development of graph neural network to predict paths during symbolic execution



Education level: bachelor

Speciality *02.03.03 «Software and Administration of Information Systems»*

Programme *CB.5162.2021 «Programming Technologies»*

Scientific supervisor:
C.Sc, docent S.V. Grigorev

Reviewer:
lead developer at “IntelKod” M. P. Kosticyn

Saint Petersburg
2025

Оглавление

Введение	4
1. Постановка задачи	6
2. Символьное исполнение	7
3. Существующие решения	14
4. Используемые технологии	16
5. Описание процесса обучения в инструменте PySymGym	18
6. Интеграция MLFLOW	19
7. Поддержка Path Condition	21
8. Эксперименты	23
8.1. Нормализация атрибутов вершин	23
8.2. Классификация состояний	24
8.3. Эксперименты с Path Condition	27
9. Результаты	30
9.1. Лучшая модель	30
9.2. Анализ результатов экспериментов	32
Заключение	35
Список литературы	36

Введение

В современном мире с ростом актуальности автоматизации различных процессов обработки данных и вычислений растет и необходимость в тестировании сложных систем с целью обнаружить ситуации с нежелательным, непредсказуемым или даже опасным исходом в зависимости от входных данных. Для решения этой задачи применяются разные подходы. Например, можно тестировать программу на случайных входных данных, используя фаззинг или ручное тестирование, но у таких подходов есть недостаток в том, что вариантов входных данных зачастую очень много и перебрать все невозможно. Из-за этого можно упустить входы, на которых программа ведет себя не так, как ожидалось.

Эту проблему решает символьное исполнение, в котором рассматриваются не конкретные значения параметров, а состояния, которые могут порождаться входным значением в зависимости от некоторых условий в коде. С его применением, обойдя все возможные пути графа потока управления, который строится по исполняемому коду, можно учесть все состояния, в которых может оказаться программа.

Из-за того, что состояний может быть очень много, а путь достижения максимального тестового покрытия в общем случае не единственный, подбор оптимальной последовательности шагов — нетривиальная задача. Существует множество подходов, оптимизирующих выбор очередного шага при символьном исполнении [13]. Среди них использование различных алгоритмов на графах, например, DFS, который позволяет хранить в памяти немного информации, но на больших графах может работать медленно, BFS, позволяющий исследовать пути параллельно, но требующий больших затрат по памяти. Также для максимизации тестового покрытия используются эвристики. Некоторые из них представлены в работах [2, 4, 10, 1, 3]. В [4], например, используется информация о расстоянии до ближайшей неисследованной вершины в вычислении вероятности, с которой шаг будет выбран.

Кроме того, для оптимизации символьного исполнения применяются методы машинного обучения [6, 11, 7, 12]. В стратегиях с их исполь-

зованием в предсказании участвуют различные свойства состояния, но не учитывается структура их связей между собой.

Графовые нейронные сети позволяют учитывать не только свойства состояний и вершин графа потока управления, но и связи между ними. Эта информация может оказаться полезной при выборе оптимальной последовательности шагов во время символьного исполнения.

Существует инфраструктура `PySymGym`¹ для обучения графовых нейронных сетей символьному исполнению. Она позволяет обучать модель, непосредственно взаимодействуя с символьной машиной. На основе этой инфраструктуры и выполнена данная работа, которая посвящена созданию графовой нейронной сети, предсказывающей пути при символьном исполнении.

¹<https://github.com/PySymGym/PySymGym> — ссылка на репозиторий инфраструктуры для обучения (дата доступа: 3 января 2024 г.).

1. Постановка задачи

Целью данной работы является создание графовой нейронной сети, предсказывающей пути при символьном исполнении. Для её достижения были поставлены следующие задачи.

- Интегрировать MLFLOW, который предоставляет набор инструментов для управления экспериментами, в существующую инфраструктуру для обучения графовых нейронных сетей PYSYMGYM.
- Провести эксперименты с нормализацией атрибутов вершин.
- Провести эксперименты с обучением с учителем в рамках решения задачи классификации состояний.
- Реализовать поддержку path condition, который является логической формулой и задает ограничения на переменные.
- Провести эксперименты с использованием path condition.

```

1 def function(x: int):
2     if x < 0:
3         y = x + 1
4         return y
5     else:
6         s = other_function(x)
7         s += 1
8         return s
9
10 def other_function(x: int):
11     if x == -5:
12         return 0
13     else:
14         return x

```

Листинг 1: Пример кода для символьного исполнения

2. Символьное исполнение

Символьное исполнение — техника статического анализа кода, подразумевающая рассмотрение всех вариантов входных данных, при которых выполняется каждая часть программы. Таким образом, определяется ее поведение на путях исполнения, соответствующих рассмотренным данным. С его помощью можно решать различные задачи, связанные с анализом кода: поиск уязвимостей и ошибок, максимизация процента тестового покрытия, проверка исполнения конкретной инструкции и т. д.

В этом разделе символьное исполнение будет описано, исходя из его реализации в VSHARP², на основе которого выполнена эта работа. Для лучшего понимания рассмотрим простой пример кода (листинг 1).

На каждом шаге анализа программы конфигурация символьной машины может быть представлена в виде графа следующим образом. Метод, который нужно протестировать, разбивается на базовые блоки, и по ним строится граф потока управления. К нему добавляются вершины, описывающие возможные состояния символьного исполнения.

²<https://github.com/VSharp-team/VSharp> — репозиторий V# (дата доступа: January 7, 2024).

Именно с помощью них и происходит исследование графа потока управления посредством их перемещения. Они характеризуются логической формулой, которая называется *path condition*. Эта формула задает ограничения, учитывая которые можно оказаться в конкретном базовом блоке кода. Формально полученный граф можно описать так:

$G = (V, E), V = V_1 \cup V_2 \cup V_3, E = E_{V_1} \cup E_{in} \cup E_{parent\_of} \cup E_{history} \cup E_{call} \cup E_{return} \cup E_{pc\_root} \cup E_{V_3}$, где

- V_1, E_{V_1} — множества вершин и ребер внутрипроцедурного графа потока управления;
- V_2 — состояния;
- V_3, E_{V_3} — множества вершин и ребер графа, построенного по path condition;
- $E_{in} = \{(u, v) \mid (u \in V_2, v \in V_1), \text{ состояние } u \text{ в базовом блоке } v\}$;
- $E_{parent\_of} = \{(u, v) \mid (u \in V_2, v \in V_2), \text{ состояние } u \text{ породило состояние } v\}$;
- $E_{history} = \{(u, v) \mid (u \in V_2, v \in V_1), \text{ состояние } u \text{ посетило базовый блок } v\}$;
- $E_{call} = \{(u, v) \mid (u, v \in V_1), \text{ базовый блок } u \text{ заканчивается вызовом метода с входной вершиной } v\}$;
- $E_{return} = \{(u, v) \mid (u, v \in V_1), \text{ блок } u \text{ закончился инструкцией возврата, и управление должно передаться в блок } v\}$;
- $E_{pc\_root} = \{(u, v) \mid u \in V_2, v \in V_3, \text{ вершина } v \text{ является корневой для логической формулы, характеризующей состояние } u\}$.

Таким образом, весь граф состоит из внутрипроцедурных подграфов, соединенными рёбрами E_{call} и E_{return} , дерева состояний, которое строится в процессе символического исполнения и логических формул, представленных в виде графов. Ребра $E_{history}$ и E_{in} соединяют вершины графа потока управления с состояниями. E_{pc_root} указывают на связь

между состояниями и корневой вершиной *path condition*. Кроме этого, вершины и рёбра некоторых типов снабжены атрибутами. А именно,

- все вершины $v \in V_1$ имеют следующие атрибуты:
 - **Id: uint** — уникальный идентификатор базового блока (вершины);
 - **InCoverageZone: bool** — находится ли вершина в целевой зоне;
 - **BasicBlockSize: uint** — размер базового блока в количестве инструкций;
 - **CoveredByTest: bool** — сгенерирован ли уже тест, покрывающий данный блок;
 - **VisitedByState: bool** — посещён ли данный блок каким-либо состоянием (блок посещён если существует хотя бы одно состояние, которое вошло в этот блок и покинуло его);
 - **TouchedByState: bool** — существует ли хотя бы одно состояние, посещавшее данный блок (возможно, оно ещё не покинуло его);
 - **ContainsCall: bool** — блок содержит какую-либо инструкцию вызова;
 - **ContainsThrow: bool** — блок содержит инструкцию типа “бросить исключение”.
- все вершины $v \in V_2$ имеют следующие атрибуты:
 - **Id: uint** — уникальный идентификатор состояния (вершины);
 - **PathConditionSize: uint** — размер *pathCondition*-а в терминах $V\#$;
 - **VisitedAgainVertices: uint** — общее количество повторно посещённых состоянием вершин;

- **VisitedNotCoveredVerticesInZone: uint** — общее количество вершин в целевой зоне, которые на момент посещения состоянием, были не покрыты тестом;
 - **VisitedNotCoveredVerticesOutOfZone: uint** — общее количество вершин вне целевой зоны, которые на момент посещения состоянием, были не покрыты тестом;
 - **StepWhenMovedLastTime: uint** — номер шага, на котором данное состояние было подвинуто (“временная метка”);
 - **InstructionsVisitedInCurrentBlock: uint** — количество инструкций, которые уже посещены состоянием в текущем базовом блоке.
- все рёбра $(u, v) \in E_{history}$ имеют следующие атрибуты:
 - **NumOfVisits: uint** — количество посещений вершины v состоянием u ;
 - **StepWhenVisitedLastTime: uint** — номер шага на котором состояние u посещало вершину v в последний раз.
 - вершины логической формулы $v \in V_3$ содержат:
 - **Id: uint** — уникальный идентификатор вершины;
 - **Type: uint** — тип терма (всего было выделено 49 типов).

Так, вышеописанный граф со всеми атрибутами далее будет называться графовым представлением конфигурации символьной машины, где конфигурация — текущее состояние символьной машины во время исследования программы.

На рис. 1 представлен граф, описывающий некоторую конфигурацию символьной машины во время исследования метода *function* из листинга 1. Граф содержит в себе все вышеописанные вершины и ребра, кроме подграфа логической формулы и ребер *call* и *return*. Они будут представлены далее.

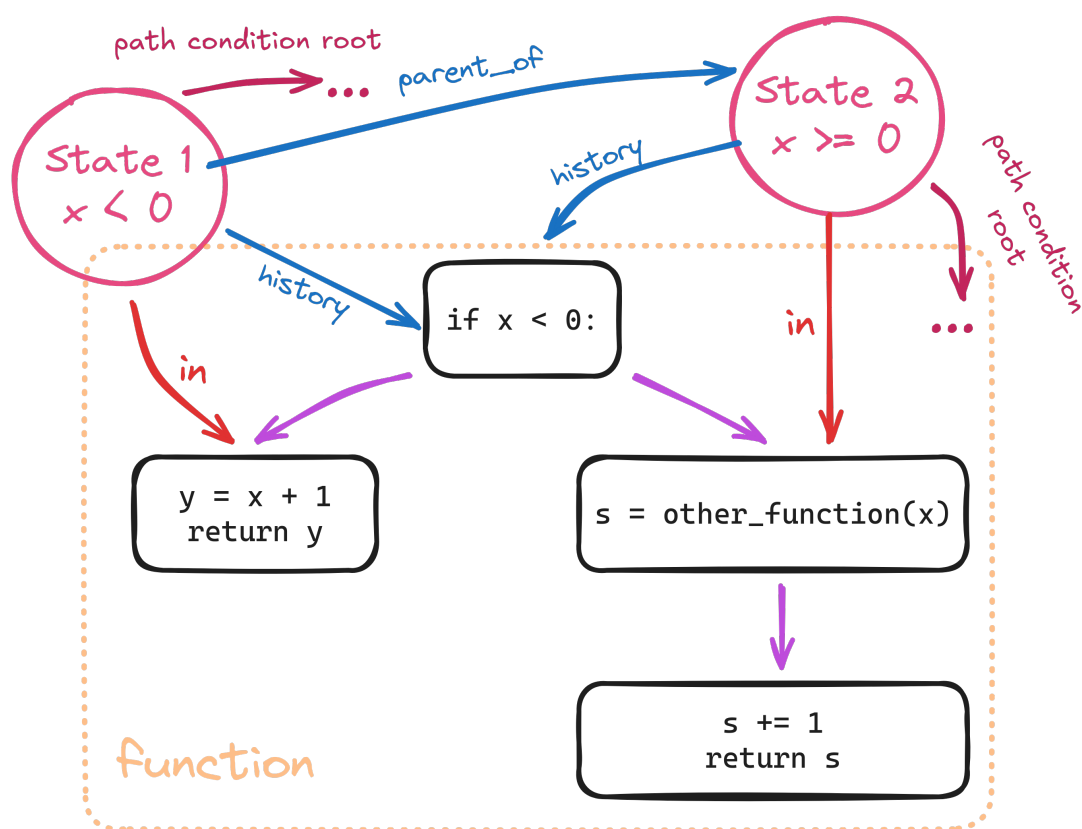


Рис. 1: Конфигурация символьной машины при исполнении кода в листинге 1

В данной ситуации после нескольких шагов, сделанных ранее, имеется два состояния, описывающих два дальнейших пути для анализа программы. Мы можем выбрать одно из них и продвинуть дальше по графу потока управления.

Предположим, что мы сделали выбор в пользу второго состояния (*State 2*). Нам нужно передвинуть его дальше по ребрам графа потока управления. В данной ситуации предыдущая вершина, в которой находилось состояние, содержала вызов функции *other_function*. В таком случае, сначала достраивается граф потока управления вызванного метода и присоединяется к существующему с помощью ребер *call* и *return*, а затем состояние перемещается в первую вершину получившегося подграфа. Полученный граф новой конфигурации представлен на рис. 2.

Теперь у нас снова есть два состояния. Предположим, что мы в очередной раз выбрали второе и получили ситуацию, представленную на рисунке 3. Так как вершина, в которой находилось это состояние

содержала условие `if x = -5`, то для второго состояния мы получили обновленную формулу `x != 5 & x >= 0`, представленную на рисунке в виде графа. Стоит отметить, что при построении графа формулы для одинаковых термов не создаются несколько отдельных вершин, а используется только одна. Так, в представленном примере переменная `x` встречается два раза, но в построенном графе ей соответствует только одна вершина.

Еще следует отметить, что после последнего описанного шага мы должны были получить еще одно состояние с условием `x = -5 & x >= 0` и связать его со вторым ребром `parent_of`, но, так как это условие не может быть выполнено для любого `x`, эта ветвь не будет дальше рассматриваться в символьном исполнении, поэтому для нее и не было создано отдельного состояния. В символьной машине выполнимость формулы определяется SMT-решателем.

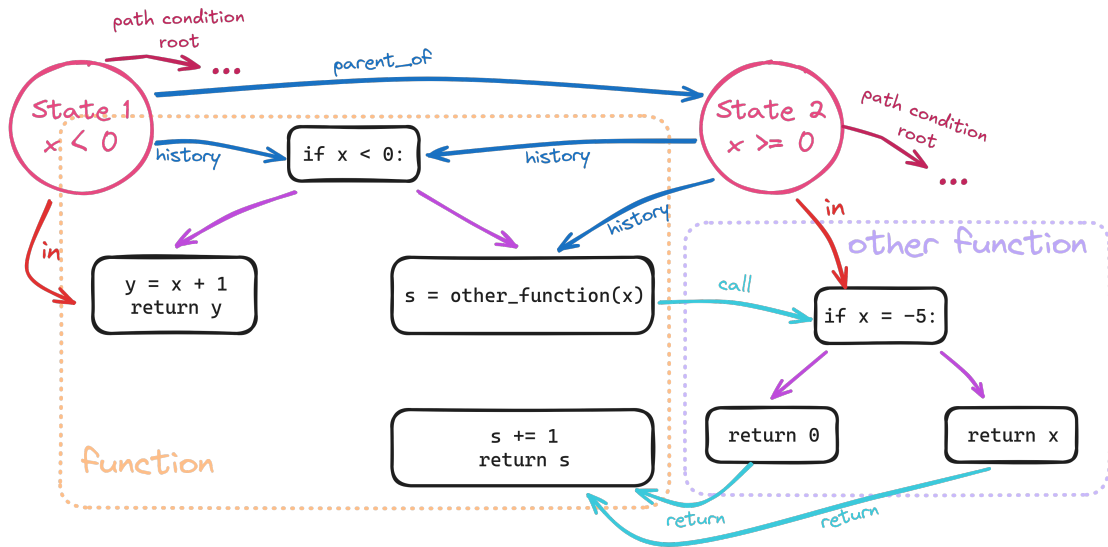


Рис. 2: Конфигурация символьной машины при исполнении кода в листинге 1, полученная при перемещении второго состояния из конфигурации на рис. 1

Таким образом, приведенным выше примером были продемонстрированы графовое представление конфигурации символьной машины и основные принципы и механизмы, возникающие во время символьного исполнения. Весь описанный процесс можно представить в виде игры, где граф потока управления — игровое поле или карта, а состояние

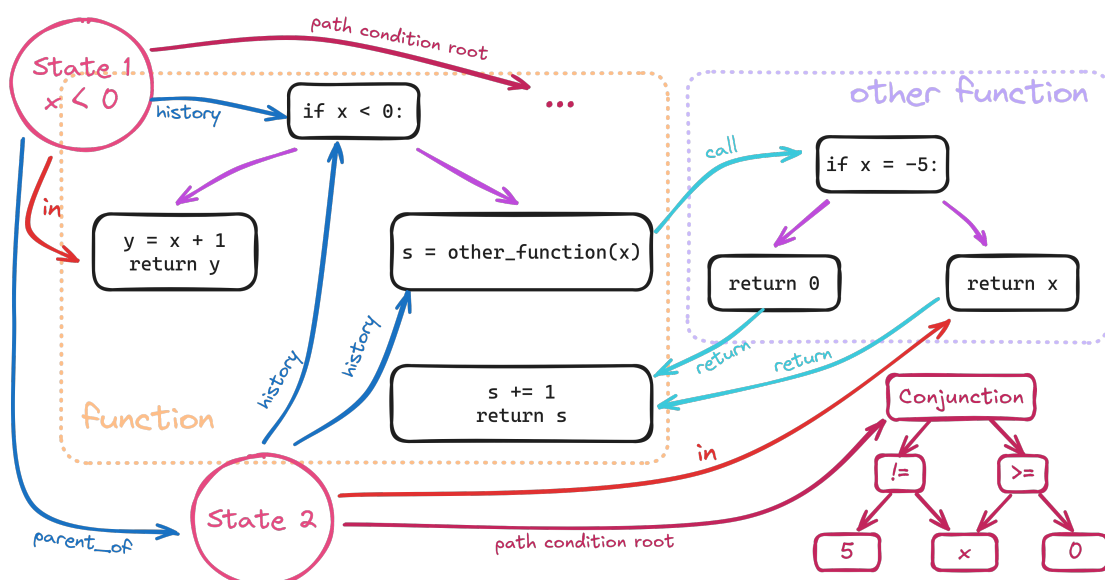


Рис. 3: Конфигурация символьной машины при исполнении кода в листинге 1, полученная при перемещении второго состояния из конфигурации на рис. 2

— фишки, которыми можно ходить. Так как символьное исполнение может применяться для решения различных задач, целей у игры может быть много. В данной работе решается задача генерации тестов с максимальным процентом покрытия за минимальное число шагов.

При дальнейшем описании проделанной работы будет использоваться приведенная игровая аналогия и описанное графовое представление конфигурации символьной машины.

3. Существующие решения

Так как проблема выбора путей в символьном исполнении является актуальной уже длительное время, существует множество решений, оптимизирующих полный перебор возможных путей для решения различных задач, связанных с анализом кода.

К оптимизации символьного исполнения были применены разные подходы. Среди них есть несколько, основанных на машинном обучении.

В LEARCH [6] для определения наиболее удачного состояния на очередном шаге используется линейная регрессия, основанная на присвоении каждому состоянию числа, которое зависит от покрытия, полученного этим состоянием. Для анализа текущей ситуации используются признаки, подобранные вручную.

FEATMAKER [12] в ходе символьного исполнения автоматически генерирует признаки для предсказания оптимального пути. Кроме того, в этой работе для генерации признаков активно используются свойства path condition.

Работа [7] посвящена применению техники активного обучения для максимизации тестового покрытия и количества найденных уязвимостей.

В работе [11] авторы применяют технику обучения с подкреплением для обеспечения прохождения исполнения через конкретную инструкцию в коде. В этом исследовании используется алгоритм *Q-learning*, который заключается в изменении таблицы состояний (*Q-table*) в соответствии с наградой, которая присваивается каждому сделанному шагу, исходя из заранее определенных правил. После чего измененная таблица используется для принятия очередного решения. Похожий подход используется в инструменте для нахождения уязвимостей в смарт-контрактах MYTHRILQL [8]. Для повышения эффективности авторы удаляют в ходе исполнения пути, про которые заранее известно, что они не могут привести к уязвимости.

В SYML [9] исследовали применимость алгоритмов классического

машинного обучения к нахождению путей с уязвимостями. В этой работе решалась задача классификации путей на варианты с уязвимостями и без них по некоторым выделенным признакам. Предполагалось, что если убрать все пути, которые выполняются корректно, то это значительно ускорит их полный перебор, так как таких вариантов исполнения программы, как правило, большинство. В LEO [5] различные алгоритмы машинного обучения применялись для изменения определенных частей исполняемого кода (применение к ним оптимизаций) для ускорения символьного исполнения.

Таким образом, машинное обучение активно применяется для оптимизации символьного исполнения и при этом показывает сравнимые, а в некоторых случаях даже лучшие результаты по сравнению с графовыми алгоритмами и различными эвристиками. В приведенных выше работах при генерации признаков, подающихся на вход модели, учитывается лишь часть особенностей конфигурации символьной машины, а так как она представима в виде графа, описанного в разделе 2, то анализ и вершин, и ребер одновременно вместе с соответствующими признаками может в значительной степени повлиять на решение задачи выбора путей при символьном исполнении, что говорит о возможности применения графовых нейронных сетей в этой области. Кроме того, в некоторых работах для генерации признаков успешно использовался *path condition*, который тоже может быть представлен в виде графа и проанализирован с помощью графовой нейронной сети.

4. Используемые технологии

Существует множество библиотек для работы с нейронными сетями. Так как в задаче подразумевается использование графовых моделей, было принято решение использовать фреймворк PYTORCH³ совместно с библиотекой PYTORCH-GEOMETRIC⁴, содержащей в себе большое количество реализаций различных обучаемых операторов. Также при выборе инструмента учитывалась подробность документации и стабильность его работы.

В обучении нейронных сетей большую роль играет подбор гиперпараметров. Для их поиска есть различные алгоритмы, а так же библиотеки с их реализациями. В ходе выбора основное внимание уделялось совместимости с доступным для обучения оборудованием и разнообразию реализованных алгоритмов. Этим требованиям удовлетворяла библиотека OPTUNA⁵.

Также при проведении экспериментов важно обеспечивать их воспроизводимость. И так как в проекте задействовано множество сторонних библиотек, было принято решение использовать менеджер зависимостей POETRY⁶ для фиксации и проверки совместимости версий пакетов. Основное его преимущество относительно использования PIP, который изначально тоже рассматривался для работы с библиотеками, заключается в возможности разделения основных зависимостей и рекуррентных, что позволяет отдельно фиксировать версии основных пакетов, и при этом автоматически подбирать совместимые рекуррентные зависимости.

Для воспроизводимости экспериментов в области машинного обучения также важно сохранять различные артефакты, логировать метрики и фиксировать гипер-параметры. Для этого есть специальные ин-

³<https://pytorch.org/> — документация PYTORCH (дата доступа: January 7, 2024).

⁴<https://pytorch-geometric.readthedocs.io/en/latest/index.html> — документация PYTORCH-GEOMETRIC (дата доступа: January 7, 2024).

⁵<https://optuna.readthedocs.io/en/stable/index.html> — документация OPTUNA (дата доступа: January 7, 2024).

⁶<https://python-poetry.org/> — документация POETRY (дата доступа: March 24, 2024).

струменты, один из них MLFLOW⁷. Он предоставляет удобный интерфейс для взаимодействия с артефактами запусков: дает возможность логировать все необходимые файлы, гипер-параметры, строит графики по сохраненным метрикам. Поэтому было решено использовать его для сохранения данных о запусках.

⁷<https://mlflow.org/> — ссылка на сайт MLFLOW (дата доступа: January 4, 2024)

5. Описание процесса обучения в инструменте PySymGym

Данное исследование было проведено на основе существующей инфраструктуры для обучения графовой нейронной сети выбору путей при символьном исполнении. На рисунке 4 представлен процесс обучения, который состоит из основного цикла (Main Loop) и двух циклов внутри него.

Первый внутренний цикл (Hyperparameters Tuning Loop) отвечает за подбор гипер-параметров и обучение с учителем в условиях имеющегося набора данных. Внутри него используется функция потерь для оценки качества модели.

Второй цикл (For Each Method) производит непосредственно символьное исполнение, используя модель и гипер-параметры, полученные на предыдущем этапе. При этом во время символьного исполнения с использованием лучшей на данный момент модели есть вероятность, что будет получена последовательность шагов, которая более оптимальна, чем уже имеющаяся. Тогда набор данных обновляется. В этом и есть главная цель второго этапа обучения. Важно отметить, что здесь качество модели оценивается четырьмя метриками: процентом тестового покрытия, числом сгенерированных тестов, количеством затраченных шагов и числом найденных ошибок. В предыдущем предложении метрики перечислены в порядке убывания их приоритета, соответственно, основной из них является процент покрытия.

После обновления набора данных старые гипер-параметры могут перестать быть актуальными, тогда нужно повторить процесс их подбора. На изображении чередование двух этапов представлено в виде цикла Main Loop.

Для обновления набора данных также сохраняются модели и гиперпараметры каждую эпоху. Помимо этого по полученным результатам создается csv файл, в котором каждой карте сопоставляется полученный результат. Из метрик логируется среднее покрытие по каждой интегрированной символьной машине, общее среднее покрытие по всем методам, количество неудачно завершившихся методов по каждой символьной машине и среднее тестовое покрытие в имеющемся на данный момент наборе данных. Для вычисления всех перечисленных метрик был создан отдельный модуль.

7. Поддержка Path Condition

Для того, чтобы добавить в граф конфигурации символьной машины path condition, нужно было реализовать его сборку и сериализацию на стороне символьной машины и PySymGym, изменить интерфейс для модели в ONNX формате, а также добавить обработку соответствующего подграфа в обучаемой модели. Модули, которые были изменены, представлены на рис. 5.

В качестве символьной машины использовался VSHARP.

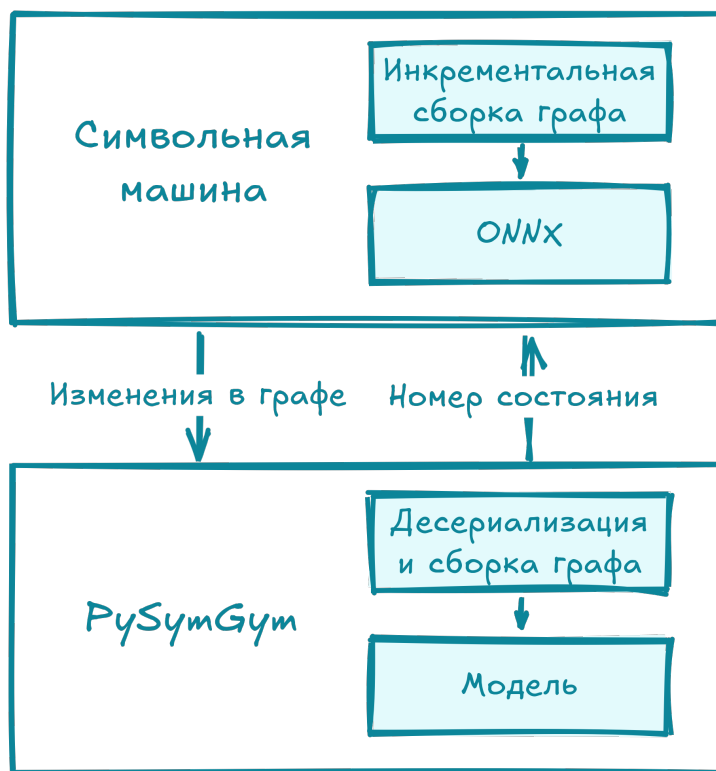


Рис. 5: Измененные для поддержки path condition модули

При преобразовании логической формулы в граф на стороне символьной машины важно было учитывать некоторые особенности.

Во-первых, некоторые части формулы могут дублироваться. Важно было реализовать сборку так, чтобы для одинаковых термов создавалась только одна вершина, и на нее одну ссылались оттуда, откуда необходимо.

Во-вторых, сборка должна была быть инкрементальной. То есть после очередного шага граф не должен был пересобирается заново, а

должны лишь были быть добавлены измененные его части. Было выяснено, что между шагами может изменяться формула у выбранного состояния. С учетом этого факта и реализовывалась сборка.

Для изменения интерфейса ONNX и десериализации на стороне инфраструктуры PYSYMGYM нужно было преобразовать вершины и ребра графа `path condition` в матрицу смежности, представленную в разреженном формате, используемом в PYTORCH-GEOMETRIC. Каждая вершина графа формулы имеет тип, который представлен числом от 0 до 48. Она преобразуется в тензор с помощью one-hot кодирования и используется в качестве признака.

У модели была изменена архитектура: добавлены новые слои для анализа нового подграфа. Подробнее новая архитектура будет рассмотрена далее.

8. Эксперименты

В ходе работы были проведены эксперименты с нормализацией атрибутов вершин и с рассмотрением процесса обучения с учителем, как решения задачи классификации состояний. В этих экспериментах не использовался граф `path condition`. В качестве характеристики формулы использовался только ее размер (`PathConditionSize`).

Затем была проведена серия экспериментов с использованием графового представления логической формулы в качестве ее характеристики вместо `PathConditionSize`, где также была изменена архитектура модели.

В данном разделе приведены подробное описание всех серий экспериментов и полученных результатов. Все эксперименты по сравнению моделей проводились с ограничением по времени в 180 секунд. В качестве функции потерь везде, кроме эксперимента с классификацией состояний, использовалась дивергенция Кульбака-Лейблера.

$$L(y_{\text{pred}}, y_{\text{true}}) = y_{\text{true}} \cdot \log \frac{y_{\text{true}}}{y_{\text{pred}}}$$

8.1. Нормализация атрибутов вершин

Как говорилось в разделе 2, у вершин графа потока управления и состояний есть атрибуты. Некоторые из них не имеют определенного диапазона значений (например, `PathConditionSize` у состояний может принимать сколь угодно большое значение). Это наталкивает на мысль о том, что может возникнуть ситуация, когда значение будет таким, что будет сильно отличаться от всех имеющихся в наборе данных, что может отрицательно сказаться на качестве работы модели. Для этого было решено провести серию экспериментов с нормализацией таких атрибутов.

Для сравнения были выбраны три функции нормализации:

- $\text{min\_max\_scaling}(x) = \frac{x - \min(x)}{\max(x) - \min(x)}$
- $l_{\infty}(x) = \frac{x}{\|x\|_{\infty}}$

- $l_2(x) = \frac{x}{||x||_2}$

Сначала нормализация была применена только к одному атрибуту состояний — `VisitedAgainVertices`. Результаты представлены на рисунке 6. Хотя на обучающей выборке никакая из функций не выделялась (рис. 6a), на валидации сильно лучше оказались *min_max_scaling* и l_∞ (рис. 6b). С ними был проведен эксперимент, в котором были задействованы все атрибуты с неизвестным диапазоном значений. Результаты представлены на рисунке 7. *min_max_scaling* оказался значительно лучше на валидации.

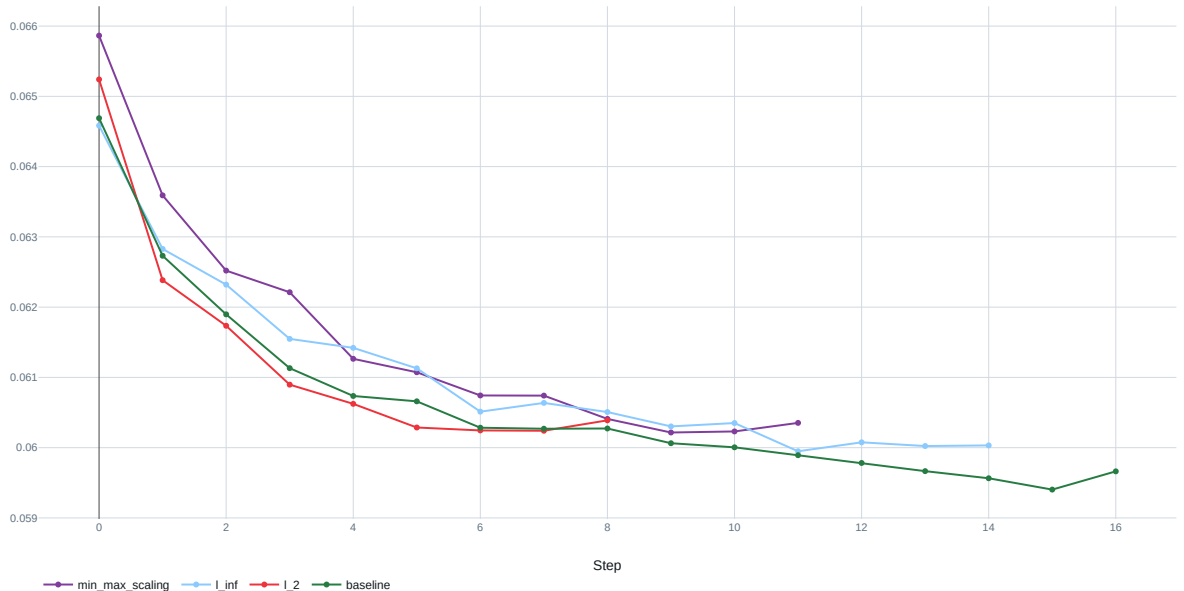
Далее было проведено сравнение модели с нормализацией с моделью без нее с их непосредственным использованием во время символьного исполнения (рис. 8). Модель с нормализацией оказалась лучше в 9 случаях хуже в 19, причем во многих из них процент покрытия получился значительно меньше.

Вывод

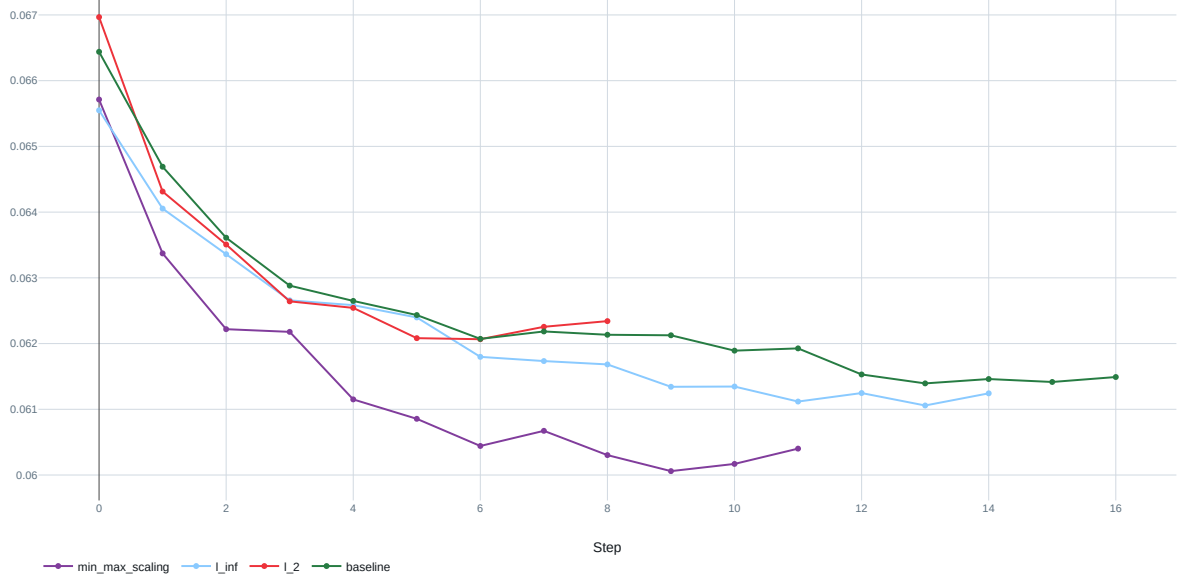
- В рамках символьного исполнения нормализация ухудшает работу нейронной сети. Вероятно, для максимизации тестового покрытия ключевую роль играют абсолютные значения атрибутов.
- Так как модель с нормализацией оказалась лучше с точки зрения функции потерь, но хуже относительно процента тестового покрытия, можно сделать вывод что эти целевые функции не соотносятся так, как ожидалось. Решению этой проблемы посвящена следующая серия экспериментов.

8.2. Классификация состояний

На задачу обучения нейронной сети символьному исполнению с точки зрения обучения с учителем можно смотреть по-разному. Все выше описанные эксперименты проводились исходя из идеи, что каждое состояние может быть выбрано с определенной вероятностью, а то, что с наибольшей вероятностью, считается самым удачным для выбора в



(а) График изменения значения функции потерь на обучающей выборке

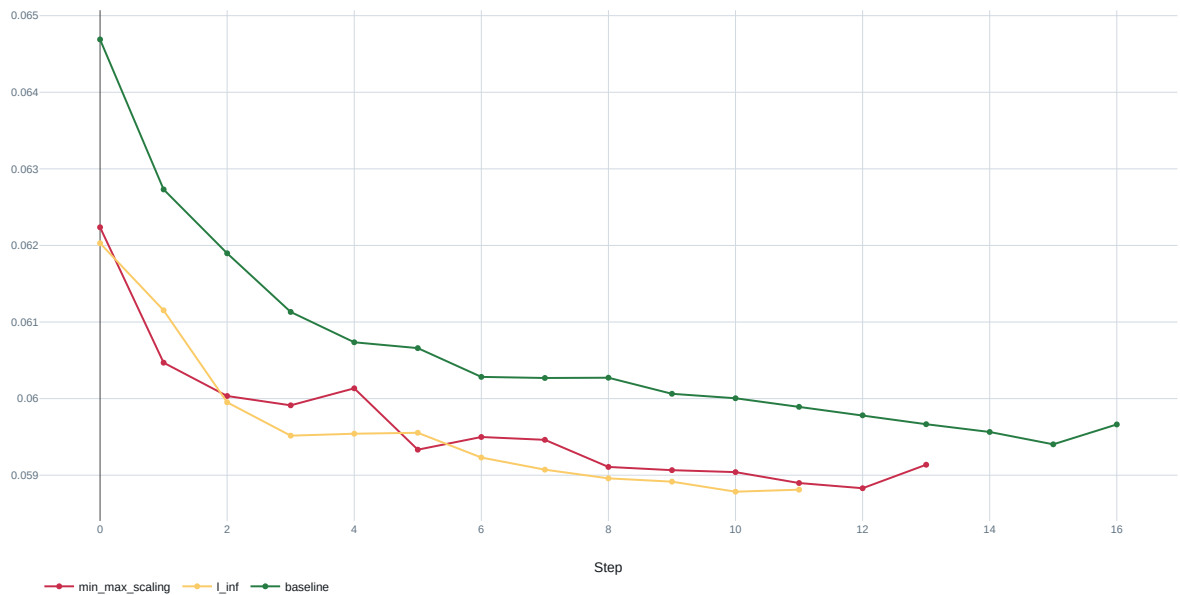


(б) График изменения значения функции потерь на валидационной выборке

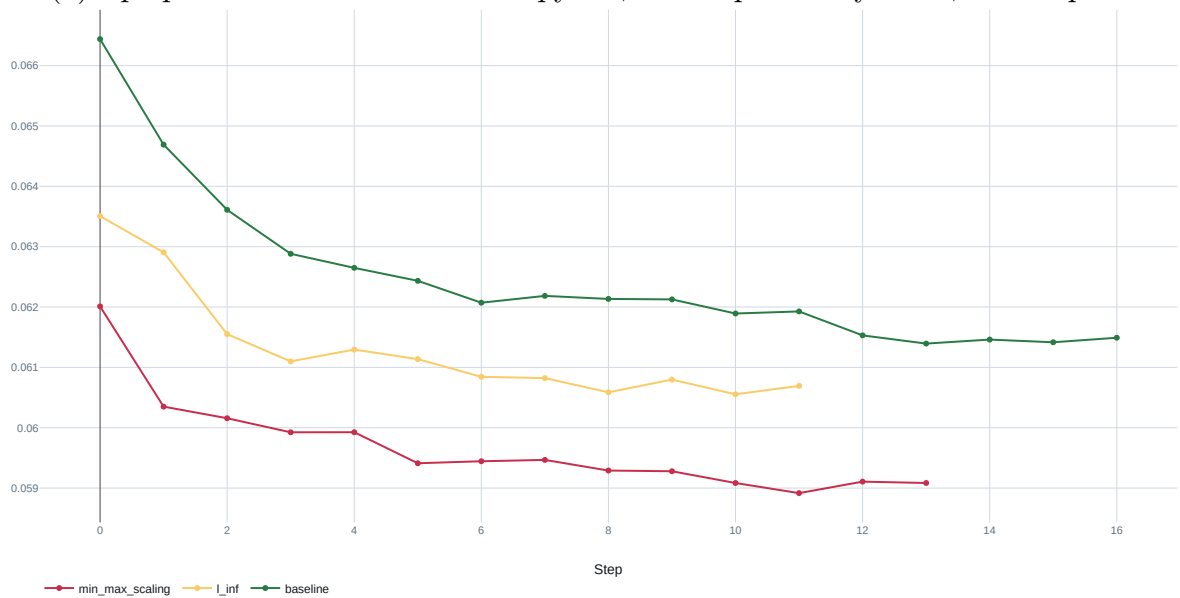
Рис. 6: Результаты эксперимента с нормализацией одного атрибута у состояний.

текущей ситуации. В этом разделе предлагается посмотреть на этот процесс с другой точки зрения, а именно считать, что состояние может принадлежать к одному из двух классов: либо его стоит выбирать с данной ситуации, либо не стоит. В качестве функции потерь в этом подходе использовалась бинарная кросс-энтропия.

$$L = \{l_1, \dots, l_N\}^\top, \quad l_n = -y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)$$



(а) График изменения значения функции потерь на обучающей выборке



(б) График изменения значения функции потерь на валидационной выборке

Рис. 7: Результаты эксперимента с нормализацией нескольких атрибутов

На рисунке 9 представлены результаты сравнения модели, обученной с использованием вероятностного подхода, с той, которая решала задачу классификации. Результаты не сильно отличаются между собой. У обеих моделей есть методы, на которых они оказались лучше, но эти числа не сильно отличаются. На основной части исполняемых методов были получены одинаковые результаты.

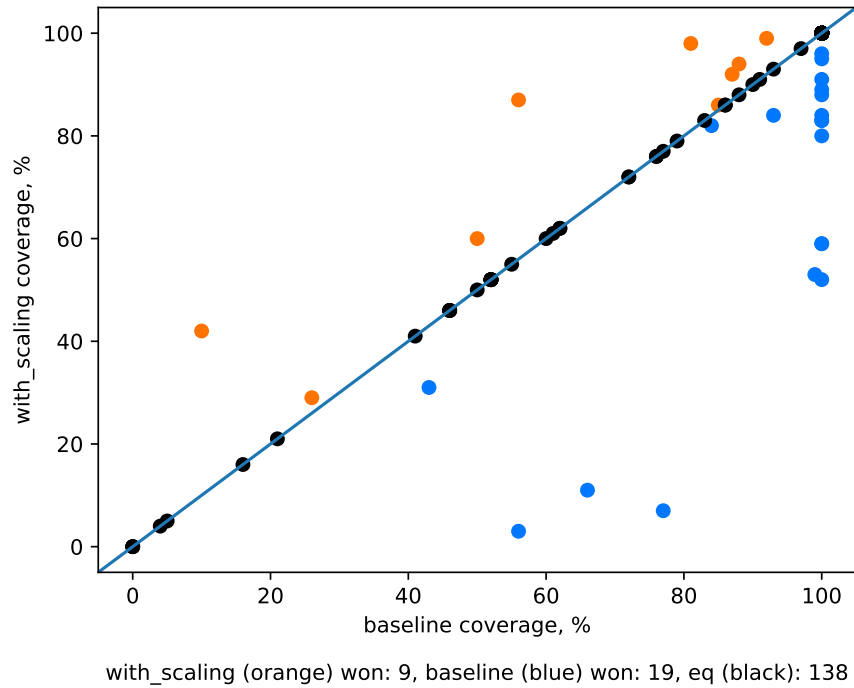


Рис. 8: Сравнение результатов модели с нормализацией с результатами модели без нее

Вывод

Эта серия экспериментов проводилась с использованием лучшей нормализации из вышеописанных экспериментов. Результаты двух подходов отличаются не существенно. Так как с предыдущим подходом было запущено больше экспериментов и собрано больше данных, было решено оставить его.

8.3. Эксперименты с Path Condition

В данной серии экспериментов помимо изменения графового представления конфигурации символьной машины, также была изменена архитектура модели для анализа нового типа вершин. Результаты экспериментов в сравнении с лучшей на данный момент моделью, которая анализирует граф без логической формулы, представлены на рисунке 10.

В данной серии экспериментов за основу была взята архитектура предыдущей модели (на графике baseline) и к ней были добавлено несколько сверточных слоев для анализа подграфа формулы. Как видно из гра-

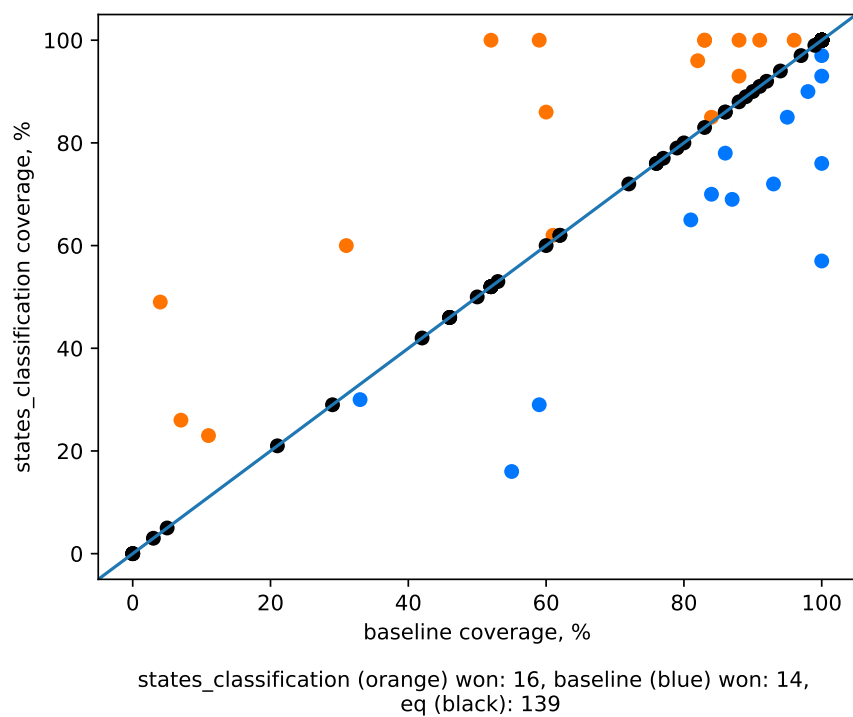


Рис. 9: Сравнение подхода, основанного на классификации состояний с вероятностным

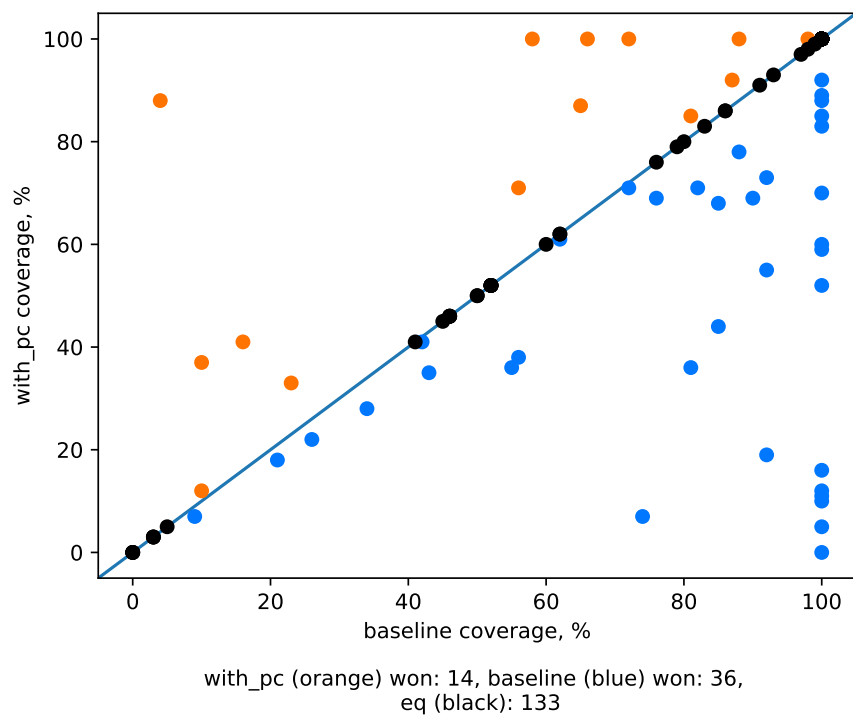


Рис. 10: Сравнение модели, анализирующей path condition, с лучшей моделью без анализа формулы

фика сравнения двух моделей по проценту тестового покрытия, результат получился сравнимый с предыдущей архитектурой, так как на большинстве карт процент модели показали одинаковый результат. Однако число методов, на котором модель без дополнительных слоев оказалась лучше превосходит количество, где лучшей была модель, анализирующая формулу. Причем на некоторых методах результаты значительно отличаются.

Причин может быть несколько. Одна из них заключается в том, что для анализа дополнительных вершин требуется время, и новая модель работает значительно медленнее старой. График сравнения нейронных сетей по времени представлен на изображении 11. В половине случаев модель без обработки формулы оказывается быстрее, соответственно, она имеет больше шансов сгенерировать больший процент покрытия за отведенное ей время.

Было проведено несколько экспериментов с упрощенной архитектурой с целью лучшего понимания работы нейронной сети, упрощения ее анализа и подбора гиперпараметров, но результаты оказались значительно хуже.

Еще одна причина может заключаться в наборе данных. При обучении новой модели с новыми типами вершин процесс получения качественного набора данных нужно было начинать с самого начала, а именно с его генерации какой-либо стратегией, чтобы в нем содержался подграф логической формулы. В итоге был получен набор данных со средним покрытием сравнимым с данными без path condition (82% с формулой и 86,2% без нее), но все же чуть менее качественный, что могло плохо сказаться на качестве модели.

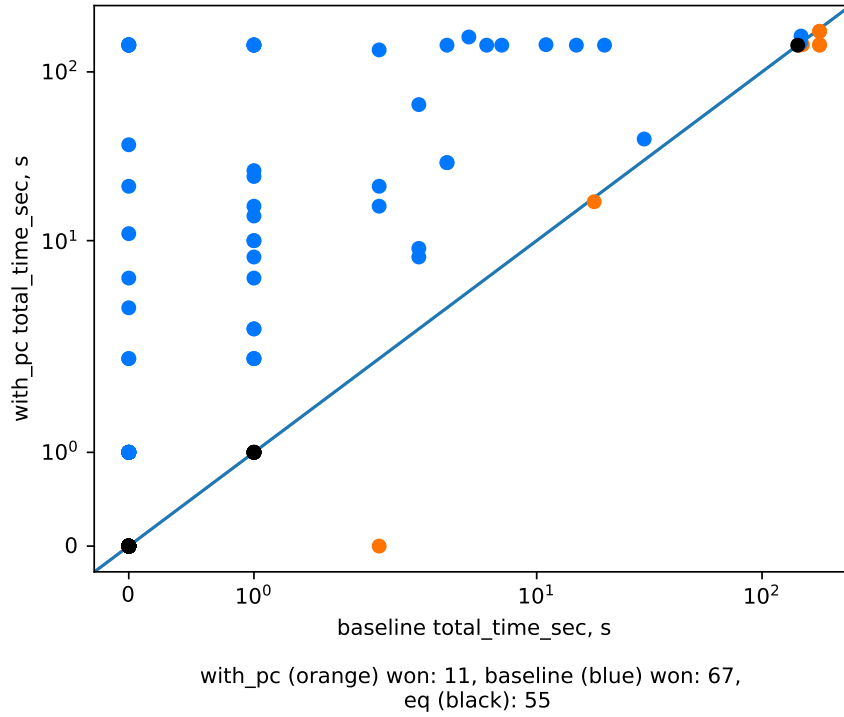


Рис. 11: Сравнение времени работы модели, анализирующей path condition, с лучшей моделью без анализа формулы при одинаковом проценте покрытия

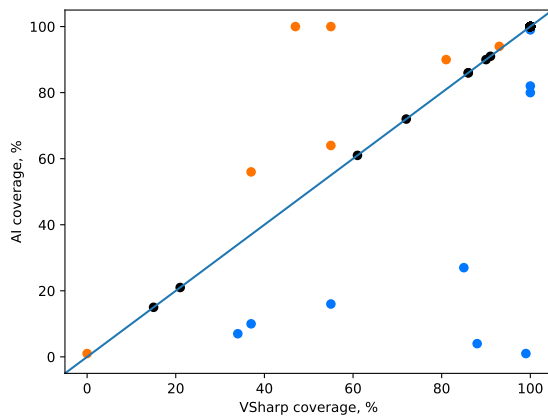
9. Результаты

В этом разделе описаны лучшие результаты, полученные на данный момент для решения задачи оптимального выбора путей при символьном исполнении для максимизации тестового покрытия при условии минимизации времени и числа сгенерированных тестов, а также направление дальнейших исследований и выводы, сделанные из экспериментов, направленных на улучшение текущих результатов.

9.1. Лучшая модель

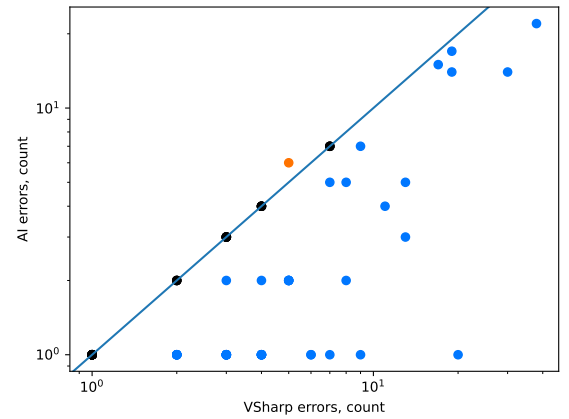
На рисунке 12 представлены результаты сравнения лучшей на данный момент нейронной сети с алгоритмом, уже имеющимся в $V\#$ и показывающим лучшие результаты относительно других внедренных стратегий. Он называется **ExecutionTreeContributedCoverage**. На всех четырех изображениях оранжевым отмечены точки, в случаях, когда обученная модель показала лучший результат, а синим в противополож-

ных ситуациях. Черный цвет соответствует одинаковым значениям.



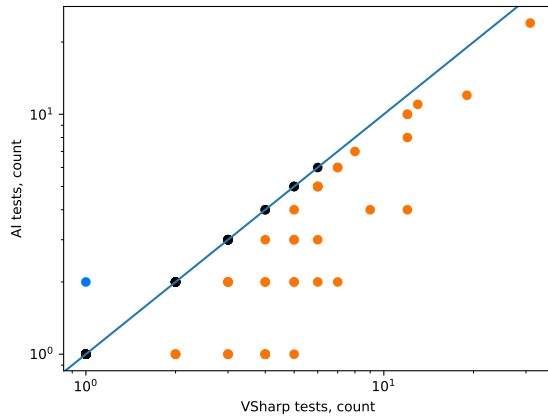
AI (оранжевый) лучше: 7, VSharp (синий) лучше: 9,
одинаковый результат (черный): 113

(а) Сравнение стратегий по полученному проценту покрытия



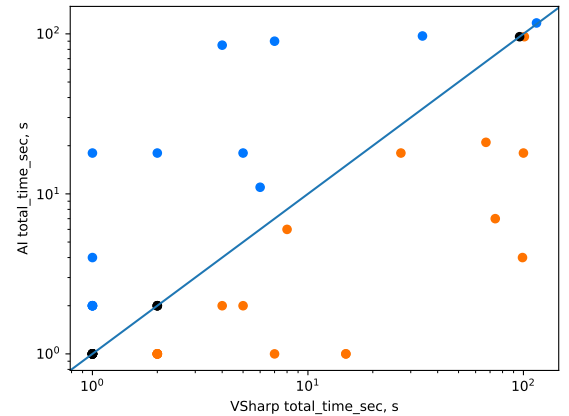
AI (оранжевый) лучше: 1, VSharp (синий) лучше: 50,
одинаковый результат (черный): 62

(b) Сравнение стратегий по количеству найденных ошибок при одинаковом проценте покрытия



AI (оранжевый) лучше: 43, VSharp (синий) лучше: 1,
одинаковый результат (черный): 69

(с) Сравнение стратегий по количеству сгенерированных тестов при одинаковом проценте покрытия



AI (оранжевый) лучше: 26, VSharp (синий) лучше: 23,
одинаковый результат (черный): 64

(d) Сравнение стратегий по времени исполнения при одинаковом проценте покрытия

Рис. 12: Результаты сравнения обученной модели с лучшей стратегией, внедренной в V#

Сравнение количества найденных ошибок, сгенерированных тестов и затраченного времени производилось на методах, где было получено одинаковое покрытие. Из-за дискретности величин некоторые точки могли накладываться друг на друга из-за чего значений на графике может оказаться меньше, чем заявлено в описании под ним.

Результаты показали, что на данный момент при сравнимых результатах по проценту покрытия и затраченному времени основным пре-

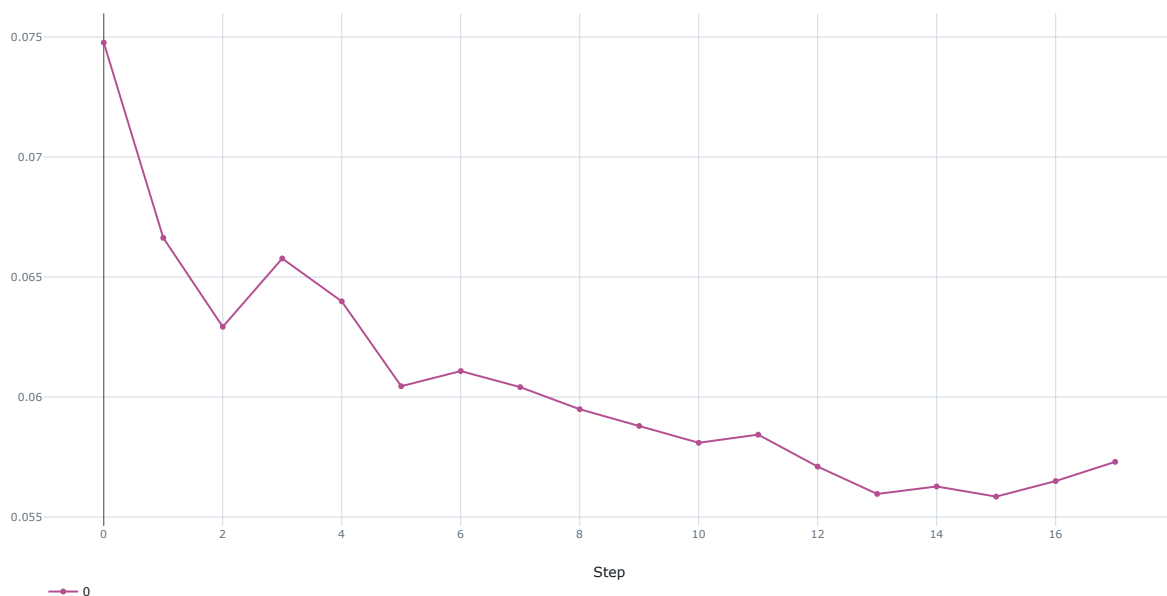
имуществом нового подхода является генерация значительно меньшего числа тестов. При этом нейронная сеть находит значительно меньше ошибок, что соответствует целевой функции обучения.

9.2. Анализ результатов экспериментов

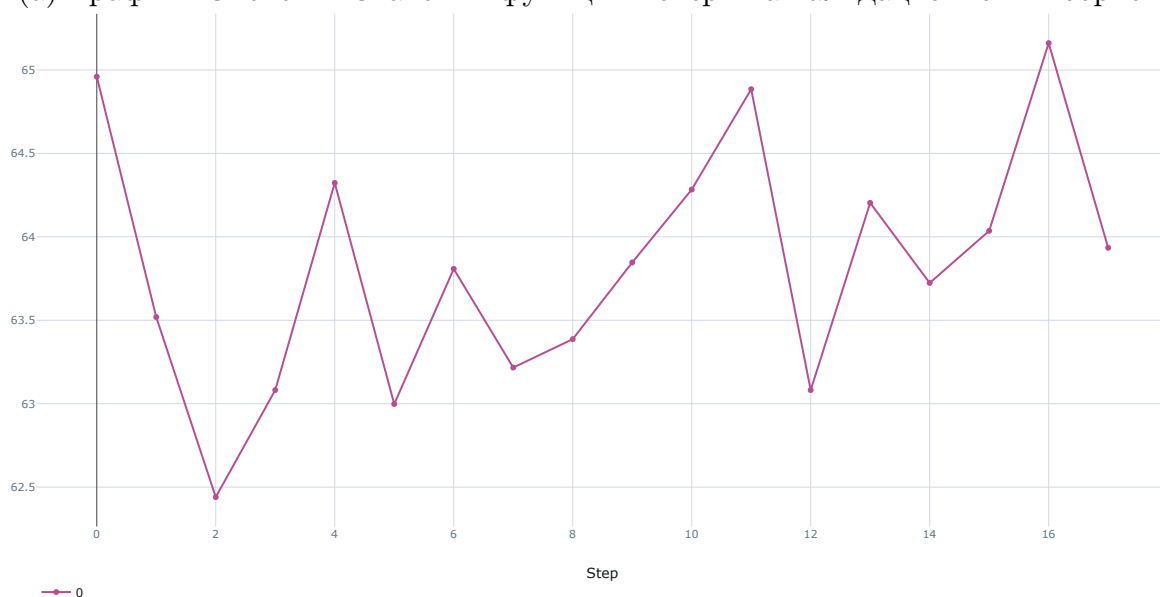
В данной работе были проведены эксперименты, нацеленные на исследование способов улучшения качества работы графовой нейронной сети во время символьного исполнения, используя подход обучения, реализованный в PYSYMGYM. Из полученных результатов можно сделать следующие выводы.

9.2.1. Согласованность метрик

Так как в подходе используются две метрики: функция потерь для обучения с учителем и процент покрытия для решения основной задачи, нужно, чтобы они согласовались между собой. Использование дивергенции Кульбака-Лейблера позволило достичь результатов, описанных в разделе 9.1, однако при детальном рассмотрении оказалось, что ее минимизация не всегда соответствует максимизации процента покрытия. На рисунке 13 приведены графики, полученные при запуске эксперимента для анализа согласованности. После каждой эпохи обучения модели запускалось две валидации: оценка дивергенции Кульбака-Лейблера и вычисление процента тестового покрытия с помощью символьной машины. Как видно, на некоторых участках графика минимизация дивергенции соответствовала максимизации покрытия, например, на промежутке с седьмой по десятую эпоху, но такое соответствие наблюдается далеко не всегда. Даже в конце эксперимента, когда модель начала переобучаться и значения на графике 13а начали расти, на изображении 13б ожидаемое значение получилось только на шестнадцатой эпохе, а на семнадцатой среднее покрытие значительно уменьшилось, хотя модель не переставала переобучаться. Таким образом, необходима серия экспериментов, направленная на подбор метрики и подхода для обучения с учителем.



(а) График изменения значения функции потерь на валидационной выборке



(б) График изменения среднего процента покрытия

Рис. 13: Результаты эксперимента по анализу согласованности метрик

9.2.2. Набор данных

При работе с набором данных в рамках данного подхода к обучению возникает несколько проблем, вызванных его постоянным обновлением. Так как он обновляется не только в тех ситуациях, когда был получен больший процент покрытия, но и когда сделано меньшее число шагов, данных с течением времени может становиться меньше, из-за чего модели их может не хватать для качественного обучения. Исключить эту

проблему можно путем расширения набора данных с помощью добавления новых методов или дублирования уже имеющихся графов.

9.2.3. Path Condition

Эксперименты с добавлением нового подграфа показали, что его анализ заметно увеличивает время работы модели, что может мешать ей получать лучшие результаты в условиях ограничения по времени. В имеющемся на данный момент наборе данных было посчитано число вершин и ребер, принадлежащих path condition и не принадлежащих ему и выяснено, что в около 99% элементов данных количество вершин и ребер логической формулы превосходит число вершин и ребер остальных типов. Устранению проблемы увеличившегося времени вычисления может помочь упрощение архитектуры модели, анализирующей граф потока управления и подграф состояний и более детальный подбор архитектуры для анализа path condition. Также в подборе архитектуры могло бы помочь исследование влияния нового подграфа на предсказание модели.

Кроме того, так как размер нейронной сети ограничен и фиксирован, в ходе ее работы будут учтены только те вершины формулы, которые находятся на определенном расстоянии от корневой, которая непосредственно соединена с состоянием. Поэтому для оптимизации анализа path condition также можно собирать его частично, в соответствии с глубиной нейронной сети: не учитывая те части, которые не будут задействованы.

Заключение

В данной работе было проведено исследование, направленное на улучшение качества предсказания путей при символьном исполнении графовой нейронной сетью.

- Интегрирован MLFLOW в инфраструктуру для обучения.
- Проведены эксперименты с нормализацией атрибутов вершин.
 - Результат с нормализацией хуже, чем без нее;
 - Функция потерь не согласуется с оптимизацией тестового покрытия.
- Проведены эксперименты с обучением с учителем в рамках решения задачи классификации состояний.
 - Результаты сравнимы с вероятностным подходом.
- Реализована поддержка path condition на стороне PYSYMGYM символьной машины VSHARP.
- Проведены эксперименты с использованием path condition.
 - Результаты сравнимы с моделью без анализа формулы;
 - Анализ path condition значительно замедляет работу модели, что необходимо учитывать при подборе архитектуры.

Исходный код PYSYMGYM доступен по [этой ссылке](#)⁸, а VSHARP— по [этой](#)⁹. Разработка велась от пользователя Anya497.

Также исследование было представлено на постерной сессии летней школы по оптимизации в Новосибирске, где одержало победу на конкурсе постеров, и, кроме того, статья по нему была представлена на воркшопе STATIC 2025 при конференции ICSE.

⁸<https://github.com/PySymGym/PySymGym> — ссылка на репозиторий PYSYMGYM (дата доступа: May 18, 2024).

⁹<https://github.com/PySymGym/VSharp> — ссылка на репозиторий VSHARP (дата доступа: May 18, 2024).

Список литературы

- [1] Chipounov Vitaly, Kuznetsov Volodymyr, Candea George. The S2E platform: Design, implementation, and applications // ACM Transactions on Computer Systems (TOCS). — 2012. — Vol. 30, no. 1. — P. 1–49.
- [2] EXE: Automatically generating inputs of death / Cristian Cadar, Vijay Ganesh, Peter M Pawlowski et al. // ACM Transactions on Information and System Security (TISSEC). — 2008. — Vol. 12, no. 2. — P. 1–38.
- [3] Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics / Sooyoung Cha, Seongjoon Hong, Jiseong Bak et al. // [IEEE Transactions on Software Engineering](#). — 2022. — Vol. 48, no. 9. — P. 3640–3663.
- [4] Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. / Cristian Cadar, Daniel Dunbar, Dawson R Engler et al. // OSDI. — Vol. 8. — 2008. — P. 209–224.
- [5] [Learning to Accelerate Symbolic Execution via Code Transformation](#) / Junjie Chen, Wenxiang Hu, Lingming Zhang et al. // 32nd European Conference on Object-Oriented Programming (ECOOP 2018) / Ed. by Todd Millstein. — Vol. 109 of Leibniz International Proceedings in Informatics (LIPIcs). — Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. — P. 6:1–6:27. — URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2018.6>.
- [6] Learning to explore paths for symbolic execution / Jingxuan He, Gishor Sivanrupan, Petar Tsankov, Martin Vechev // Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. — 2021. — P. 2526–2540.

- [7] [Path Exploration Strategy for Symbolic Execution based on Multi-strategy Active Learning](#) / Lianying He, Dalin Zhang, Dongqing Zhu et al. // Proceedings of the 15th Asia-Pacific Symposium on Internetworkware. — Internetworkware '24. — New York, NY, USA : Association for Computing Machinery, 2024. — P. 165–168. — URL: <https://doi.org/10.1145/3671016.3671403>.
- [8] [Reinforcement Learning Guided Symbolic Execution for Ethereum Smart Contracts](#) / Meng Wang, Weiliang Fei, Miao Wang, Jin Cui // 2023 30th Asia-Pacific Software Engineering Conference (APSEC). — 2023. — P. 91–100.
- [9] [SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning](#) / Nicola Ruaro, Kyle Zeng, Lukas Dresel et al. // Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses. — RAID '21. — New York, NY, USA : Association for Computing Machinery, 2021. — P. 456–468. — URL: <https://doi.org/10.1145/3471621.3471865>.
- [10] Unleashing mayhem on binary code / Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley // 2012 IEEE Symposium on Security and Privacy / IEEE. — 2012. — P. 380–394.
- [11] Wu Jie, Zhang Chengyu, Pu Geguang. Reinforcement learning guided symbolic execution // 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) / IEEE. — 2020. — P. 662–663.
- [12] Yoon Jaehan, Cha Sooyoung. FeatMaker: Automated Feature Engineering for Search Strategy of Symbolic Execution // [Proc. ACM Softw. Eng.](#) — 2024. — . — Vol. 1, no. FSE. — 22 p. — URL: <https://doi.org/10.1145/3660815>.
- [13] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.