Санкт-Петербургский государственный университет

Кафедра Системного Программирования

Группа 21.Б09-мм

Разработка инфраструктуры для обучения с учителем искусственной нейронной сети, выбирающей пути при символьном исполнении

Чистякова Анна Артуровна

Отчёт по учебной практике
в форме «Эксперимент»

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

Санкт-Петербург
2024

Оглавление

Введение

В современном мире с ростом актуальности автоматизации различных процессов обработки данных и вычислений растет и необходимость в тестировании таких 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 [12]. 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]. В стратегиях с их использовани-

ем для предсказания используются 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. Исследованию их применимости в этой области и посвящена данная работа.

1. Постановка задачи

Целью данной работы является исследование применимости графовых нейронных сетей к нахождению путей при символьном исполнении. Для её достижения были поставлены следующие задачи.

1. Разработка инфраструктуры для обучения с учителем графовой нейронной сети, включающую в себя подготовку датасета, процесс обучения и валидации. Обеспечение ее взаимодействия с другими компонентами проекта.

2. Проведение экспериментов с различными гиперпараметрами с целью достижения максимально возможного процента тестового покрытия, оценка результатов и их сравнение с аналогами.

```
def function(x: int):
    if x > 0:
        y = x + 1
        return y
    else:
        s = other_function(x)
        s += 1
        return s

def other_function(x: int):
    if x == 5:
        return 0
    else:
        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, соответствующим рассмотренным данным.

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

Метод, который нужно протестировать, разбивается на базовые блоки, и по ним строится граф потока управления. Так же к нему добавляются вершины, описывающие возможные состояния. Формально полученный граф можно описать так:

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

- V_1, E_{V_1} — множества вершин и ребер внутрипроцедурного графа потока управления;

[1] https://github.com/VSharp-team/VSharp — репозиторий V# (дата доступа: 7 января 2024 г.).

- V_2 — вершины, описывающие состояния;
- $E_{in} = \{(u, v) \mid (u \in V_2, v \in V_1),$ состояние u находится в базовом блоке $v\}$;
- $E_{parent\_of} = \{(u, v) \mid (u \in V_2, v \in V_2),$ состояние u породило состояние $v\}$;
- $E_{history} = \{(u, v) \mid (u \in V_2, v \in V_1),$ состояние u когда-либо находилось в вершине $v\}$;
- $E_{call} = \{(u, v) \mid (u, v \in V_1),$ блок u закончился инструкцией вызова метода, точкой входа которого является блок $v\}$;
- $E_{return} = \{(u, v) \mid (u, v \in V_1),$ блок u закончился инструкцией возврата, и управление должно передаться в блок $v\}$.

Таким образом, весь граф состоит из внутрипроцедурных подграфов, соединенными рёбрами E_{call} и E_{return} и дерева состояний, которое строится в процессе символьного исполнения. Ребра $E_{history}$ и E_{in} соединяют вершины графа потока управления с состояниями. 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.

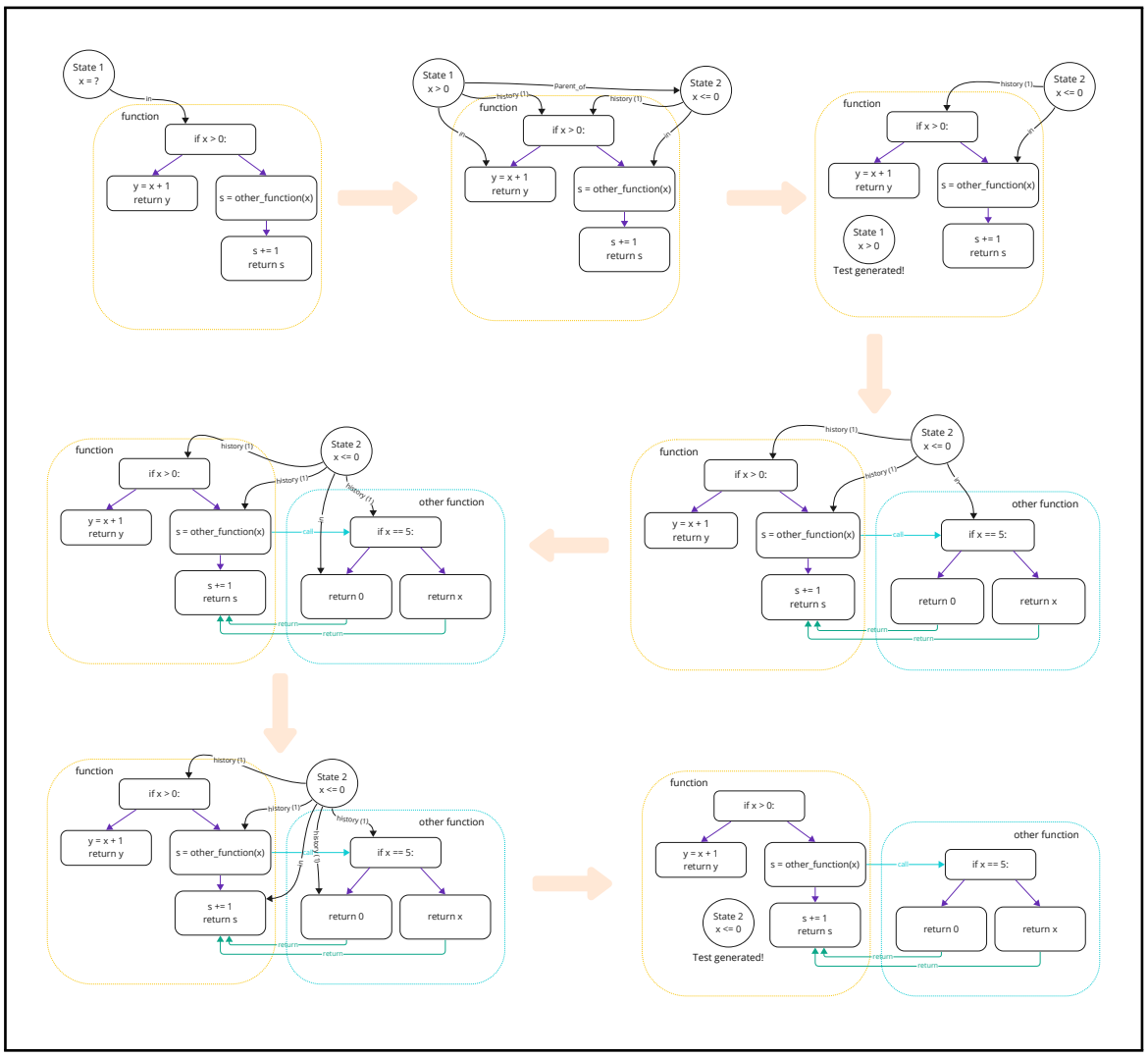


Рис. 1: Пример символьного исполнения кода в листинге 1

На рис. 1 представлен процесс символьного исполнения рассматриваемого кода. Обе функции были разделены на базовые блоки. Так как вызов *other_function* находится не в начале исполнения, первые несколько шагов общий граф не содержит подграф этого метода. Он добавится тогда, когда будет выполнена инструкция вызова этой функции.

Весь процесс символьного исполнения можно представить в виде игры, где граф потока управления — игровое поле или карта, а состояние — фишки, которыми можно ходить. Цель игры — сгенерировать тесты

с максимальным процентом покрытия за минимальное число шагов.

На изображении 1 видно, что сначала существует только одно состояние. Им мы и делаем первый шаг, который порождает второе, так как было пройдено условие на переменную x. Теперь мы рассматриваем случаи, когда $x > 0$ и $x \leq 0$. Сделаем шаг первым состоянием, так как оно находится в блоке, где программа завершается, а значит, после него будет сгенерирован тест и получен некоторый процент покрытия. Дальше мы двигаем второе состояние и нам открывается новая часть карты, соответствующая вызванному в предыдущем блоке методу. Затем продолжаем делать шаги, пока не будет получено 100-процентное тестовое покрытие.

При дальнейшем описании проделанной работы будет использоваться приведенная игровая аналогия и граф потока управления в представленном выше виде.

3. Существующие решения

Так как задача выбора путей в символьном исполнении является актуальной уже длительное время, существует множество решений, оптимизирующих полный перебор возможных путей и максимизирующих при этом тестовое покрытие.

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.

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 [7]. Для повышения эффективности авторы удаляют в ходе исполнения пути, про которые заранее известно, что они не могут привести к уязвимости.

В SyML [8] 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 уско-

рения 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 состояния символьной машины, а так как оно представимо 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.

4. Используемые технологии

Существует множество библиотек для работы с нейронными сетями. Так как в задаче подразумевается использование графовых моделей, было принято решение использовать фреймворк PyTorch[2] совместно с библиотекой PyG[3], содержащей в себе большое количество реализаций различных обучаемых операторов. Также при выборе инструмента учитывалась подробность документации.

В обучении нейронных сетей большую роль играет подбор гиперпараметров. Для их подбора 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 optuna[4].

Также при проведении экспериментов важно обеспечивать их воспроизводимость. И так как в проекте задействовано множество сторонних библиотек, было принято решение использовать менеджер зависимостей Poetry[5] 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.

[2] https://pytorch.org/ — документация PyTorch (дата доступа: 7 января 2024 г.).

[3] https://pytorch-geometric.readthedocs.io/en/latest/index.html — документация PyG (дата доступа: 7 января 2024 г.).

[4] https://optuna.readthedocs.io/en/stable/index.html — документация optuna (дата доступа: 7 января 2024 г.).

[5] https://python-poetry.org/ — документация Poetry (дата доступа: 24 марта 2024 г.).

5. Описание проекта

Проект, на основе которого реализовывалось описываемое решение, состоит из двух частей: сервер, который непосредственно производит шаги символьного исполнения кода, и клиент, который по ситуации, полученной от сервера, вычисляет следующий шаг и передает эту информацию обратно (рис. 2). При этом для ускорения взаимодействия между обеими компонентами передается не весь граф, а только часть, которая была изменена при совершении очередного шага. Сначала на сервере строится граф по исполняемому коду, затем он передается клиенту, и в последствии весь период исполнения в обеих частях хранится весь граф, который актуализируется в процессе символьного исполнения кода: с сервера передаются изменения в графе, появившиеся после очередного шага, а с клиента — номер состояния, полученный обучаемой моделью, которым нужно сделать шаг.

Рис. 2: Схема взаимодействия клиента с сервером

Клиентская часть состоит из модуля, осуществляющего взаимодействие с сервером и модуля обучения модели, который, в свою очередь, состоит из нескольких компонент, отвечающих за разные этапы обучения. Разработке второго и проведению с его помощью экспериментов и посвящена данная работа, поэтому о них далее и пойдет речь.

6. Архитектура

Для обучения было реализовано четыре модуля, представленных на рисунке 3. Dataset отвечает за подготовку, изменение и использование набора данных. Training изменяет параметры модели с помощью градиентного спуска. Validation оценивает качество модели и сохраняет результаты. Было реализовано два различных способа оценки качества нейронной сети: `validate_coverage`, подразумевающий взаимодействие с символьной машиной и вычисляющий процент покрытия и `validate_loss`, проверяющий, насколько принятые моделью решения соответствуют сохраненным в наборе данных. Также был реализован модуль Runner, запускающий обучение с заданными настройками.

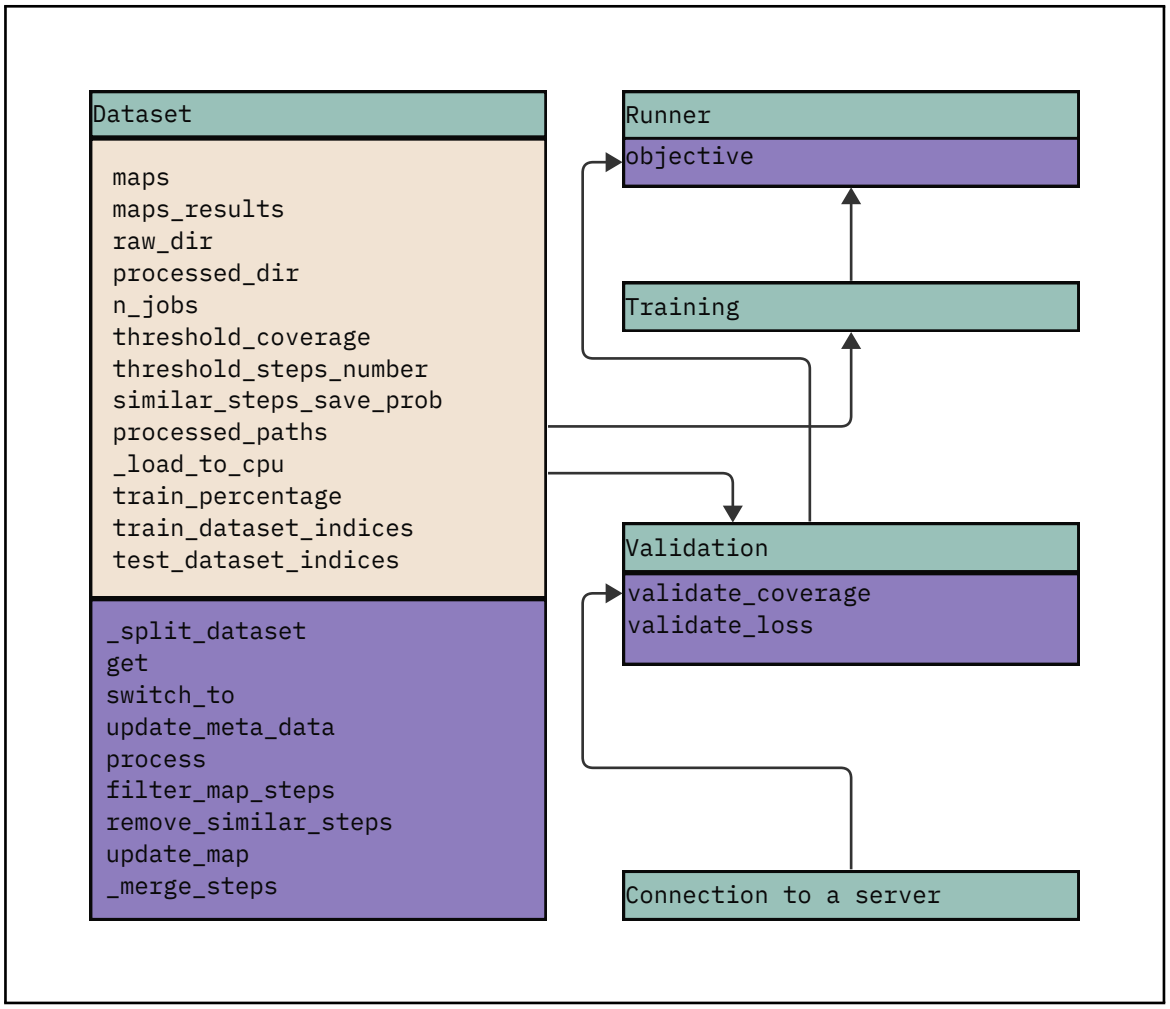


Рис. 3: Архитектура инфраструктуры для обучения

Наличие двух вариантов валидации выделило два различных сцена-

рия обучения. На основе их комбинирования были проведены эксперименты. Рассмотрим обе части подробнее.

В обоих вариантах есть совпадающие части, такие как генерация набора данных, его структура и способы обработки. Начнем с их описания.

6.1. Набор данных

Структура датасета представлена на рисунке 4. Набор данных состоит из множества методов, включающих в себя последовательность шагов и соответствующий ей результат. Каждый шаг — пара, состоящая из графа, описанного в разделе 2, и вектора, в котором каждому состоянию сопоставляется вероятность быть выбранным для очередного хода в данной ситуации.

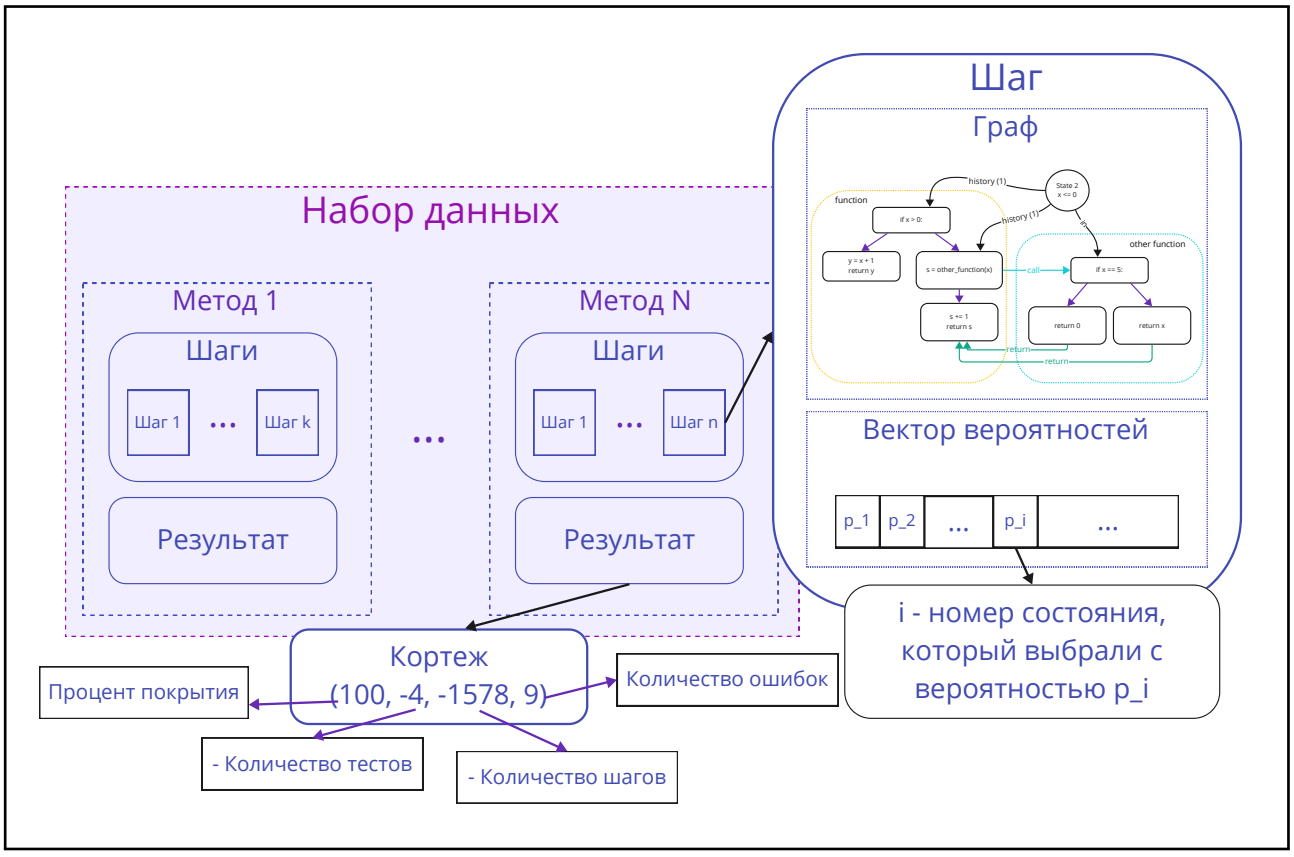


Рис. 4: Структура набора данных

Необходимый элемент для работы с набором данных — его описание, которое представляет собой файл в формате `json` с информацией о каж-

дом методе, который необходимо символьно исполнить и в дальнейшем использовать для обучения. Структура файла представляет собой список из элементов со следующими полями:

- `MapName` — уникальное имя карты;
- `DefaultSearcher` — алгоритм, которым нужно сделать первые `StepsToStart` шагов;
- `StepsToStart` — количество шагов, которое нужно сделать алгоритмом `DefaultSearcher`;
- `NameOfObjectToCover` — имя метода в исходном коде;
- `AssemblyFullName` — имя библиотеки, содержащей метод `NameOfObjectToCover`;
- `StepsToPlay` — максимальное число шагов, которое может быть совершено на данном методе.

Один и тот же метод мог встречаться несколько раз в наборе с применением различных алгоритмов (`DefaultSearcher`), которые использовались для совершения первых шагов, количество которых также варьировалось (`StepsToStart`). Так было сделано для того, чтобы разнообразить данные состояниями, в которые, вероятно, невозможно попасть, если использовать нейронную сеть.

На первом этапе работы с данными происходит их генерация. Берутся методы, описанные в `json` файле, и исполняются алгоритмом, заданным в символьной машине по умолчанию, с сохранением каждого шага в формате `json`. Также для каждого метода сохраняется полученный результат, состоящий из четырех чисел: процент покрытия, количество сгенерированных тестов, количество сделанных шагов и число найденных ошибок. Затем шаги преобразуются из формата `json` в `HeteroData`, который совместим с библиотекой, используемой для обучения.

Процесс преобразования занимает много времени, поэтому была добавлена возможность запускать его параллельно. Атрибут `n_jobs` соот-

ветствует количеству одновременно запущенных процессов. Такое преобразование происходит только при первом запуске, а в последующих используются уже сохраненные данные.

В проведении экспериментов использовалось два различных сценария обучения. Дальнейшие этапы обучения различаются в зависимости от того, какой из них был запущен. Рассмотрим отдельно каждый из них.

6.2. Сценарий 1: минимизация функции потерь

Цель данного варианта обучения заключалась в подборе гиперпараметров с использованием библиотеки OPTUNA. Запуск обучения с очередным набором параметров был выделен в отдельную функцию `objective`. Изменение параметров модели осуществлялось градиентным спуском, а затем оценивалось с помощью функции `validate_loss`. В этом варианте необходимо было разделить данные на тренировочную и валидационную выборки, поэтому в классе Dataset были реализованы возможности разделения данных на выборки (`_split_dataset`) и переключения между ними (`switch_to`).

Важно отметить, что данные в этом подходе никак не изменялись от эпохи к эпохе в отличие от сценария, который будет описан далее. Поэтому была добавлена настраиваемая возможность в начале процесса загружать шаги сразу на CPU (`Dataset._load_to_cpu`). То есть во время обучения очередной батч подгружался из оперативной памяти, а не напрямую с диска, что ускоряло использование данных. То же самое происходило и во время валидации модели.

6.3. Сценарий 2: максимизация покрытия

Этот вариант обучения заключался в улучшении качества набора данных и оценки модели непосредственно в роли оракула при символьном исполнении. Обучение нейронной сети производится так же градиентным спуском, но данные подгружаются уже не из оперативной памяти, а с диска, так как функция `validate_coverage` подразу-

мевает изменение набора данных (`Dataset.update_map`). Дело в том, что при очередном изменении весов нейронной сети, "новая" модель при игре на очередной карте может получить либо результат, лучший, чем сохраненный на данный момент, либо худший, либо абсолютно такой же. Результат — кортеж из четырех чисел: процент покрытия, $(-1)*$количество сгенерированных тестов, $(-1)*$количество сделанных шагов, число найденных ошибок. Сравнение производилось на основе лексикографического порядка (лучший результат тот, который больше). При первом исходе все сохраненные шаги, относящиеся к сыгранной карте, заменяются на новые, с лучшим результатом. Во втором варианте данные не изменяются. Основной интерес представляет третий случай. Рассмотрим его подробнее.

Если после очередной игры результат совпал с уже сохраненным, новые шаги добавляются к существующим (`Dataset._merge_steps`). При этом элементы в обоих наборах попарно сравниваются для того, чтобы найти входные графы с одинаковым набором состояний и вершин графа потока управления. То есть графы считаются одинаковыми, если в нем совпадают наборы вершин. Набор вершин считается совпадающим, если равны отсортированные в лексикографическом порядке массивы с признаками вершин. Такое сравнение имеет смысл, потому что с каждым сделанным шагом признаки меняются (это понятно из их смысла). Таким образом, если набор вершин с признаками совпал, то это одно и то же состояние символьной машины. Для таких шагов эталонные распределения складываются и нормализуются.

Этот план действий был разработан исходя из соображения о том, что оптимальный набор шагов в символьном исполнении может быть не единственным. Например, в рассмотренном на рисунке 1 случае после первого шага можно всегда выбирать второе состояние, пока оно не дойдет до конца программы, а потом уже сгенерировать тест первым. Тогда мы все еще получим 100-процентное покрытие с тем же количеством тестов и шагов, но набор принятых решений будет другим.

Процесс валидации с использованием символьной машины занимает много времени, поэтому была добавлена возможность его параллельно-

го запуска (рис. 5). Сначала задается количество процессов k, затем запускается k символьных машин и каждой из них по мере освобождения отправляются задачи с еще не сыгранными картами.

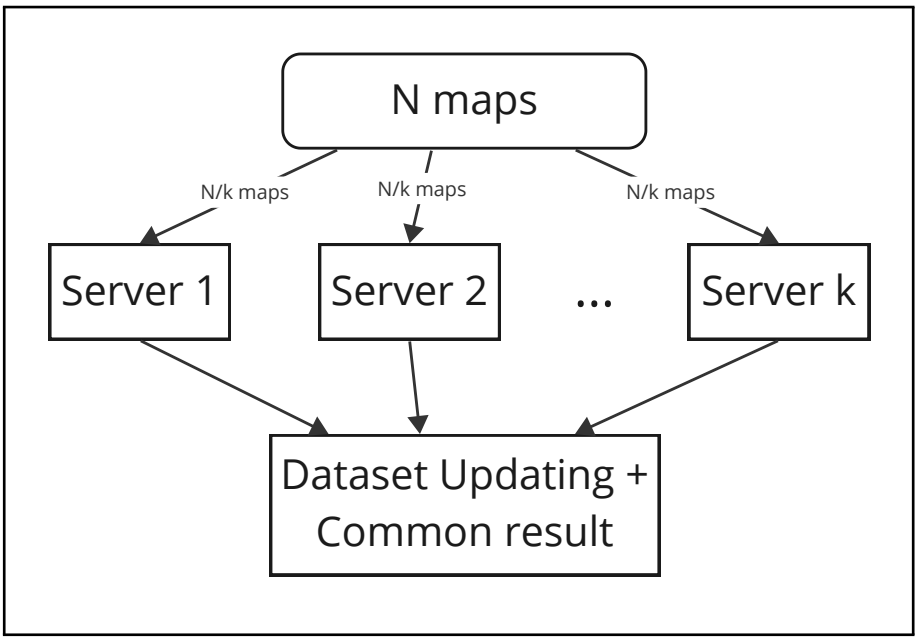


Рис. 5: Параллельность валидации

7. Эксперименты

В ходе работы были проведены эксперименты с различными архитектурами предобученных моделей, с нейронной сетью, инициализированной случайными параметрами, а также с отдельным обучением последнего слоя предобученной нейронной сети. Использовался оптимизатор Adam. Разница между полученным и эталонным распределениями оценивалась расстоянием Кульбака-Лейблера, которое вычисляется по следующей формуле 1.

$$L(y_{pred}, y_{true}) = y_{true} * (\log y_{true} - \log y_{pred}) \tag{1}$$

Набор данных состоял из 507 реализаций алгоритмов и методов различной сложности и размера в плане количества возможных состояний[6]. Среди них были простые рекуррентные алгоритмы на графах, сортировки, методы, эмулирующие работу процессоров и других устройств. Также в выборке содержались синтетические методы с большой концентрацией условий и циклов.

Проведение экспериментов представляло собой чередование запусков двух сценариев обучения. Целью первого был подбор гиперпараметров модели с помощью Tree-structured Parzen Estimator, а второго — улучшение качества набора данных с помощью лучшей модели, полученной предыдущим шагом.

В первом варианте запуска подбирались параметры обучения: размеры шага (*lr*) и батча (*batch_size*), а также параметры архитектуры модели, представленной на рисунке 6. У нейронной сети анализировалось число "прыжков" для анализа соседей в слоях TAGConv [9] (*num_hops_1*, *num_hops_2*), число признаков состояния перед применением последнего линейного слоя (*num_of_state_features*) (или размерность эмбеддинга) и количество каналов модели (*hidden_channels*). На данный момент лучшие результаты были получены со следующими значениями:

[6] https://github.com/gsvgit/PySymGym/blob/dev/maps/DotNet/Maps/dataset.json — файл с описанием набора данных, используемого в экспериментах (дата доступа: 1 июня 2024 г.).

- $lr = 0.0003$;
- $batch\_size = 109$;
- $num\_hops\_1 = 5$;
- $num\_hops\_2 = 4$;
- $num\_of\_state\_features = 30$;
- $hidden\_channels = 110$.

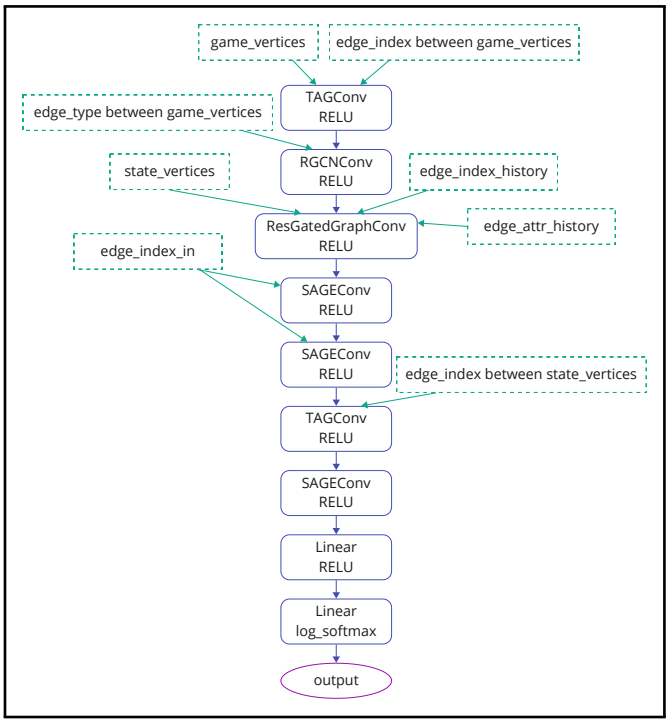


Рис. 6: Архитектура модели

Число эпох в экспериментах было фиксированным, поэтому в ходе анализа динамики изменения значений гиперпараметров было замечено, что основным при подборе являлся размер шага (рис. 7). В дальнейшем планируется провести эксперименты не с заранее заданным числом эпох, а с механизмом, останавливающим обучение по некоторому условию, что, вероятно, поможет точнее подобрать остальные гиперпараметры.

Данный сценарий обучения запускался на наборах шагов, с помощью которых достигалось 100-процентное покрытие. Несмотря на то,

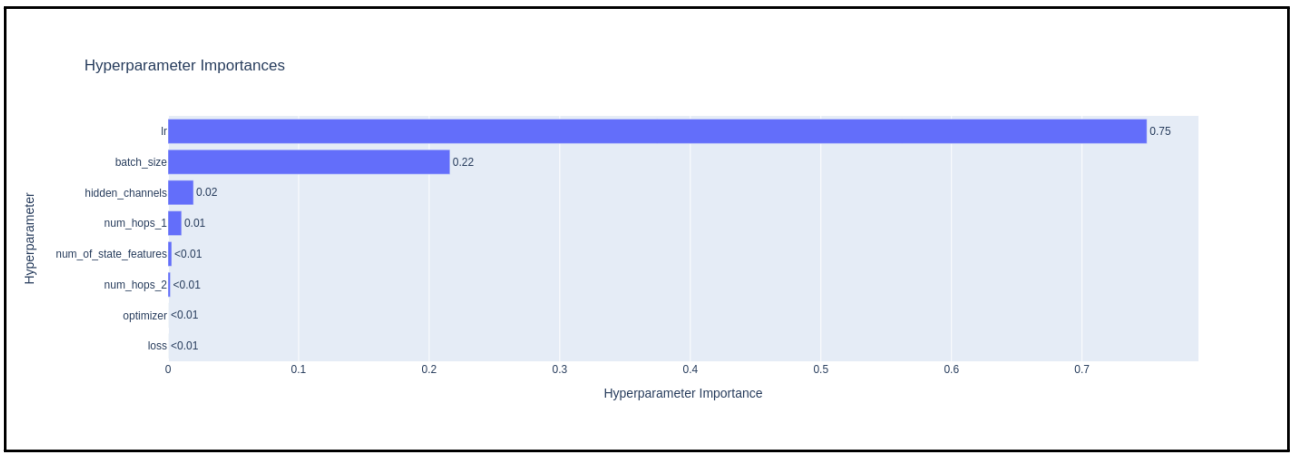


Рис. 7: Степень влияния гиперпараметров на автоматический подбор

что в датасете также хранились последовательности, с которыми был получен меньший процент, их было решено не использовать из-за того, что набор шагов для получения лучшего результата мог значительно отличаться от уже имеющегося. Поэтому использование последовательностей, в результате которых не было получено 100-процентное покрытие, могло помешать получить лучший результат.

Второй сценарий обучения с использованием символьных машин запускался с целью улучшения набора данных. В нем использовались все 507 методов, описанных в `json` файле. После этого этапа к датасет для первого способа обучения изменялся. К нему могли добавиться методы, для которых была найдена последовательность, с которой достигалось 100-процентное покрытие, также на некоторых картах могло уменьшится число шагов и т. д. В результате нескольких итераций запуска этого процесса был получен набор данных, информация о количестве карт со всеми возможными процентами покрытия которого представлена на рисунке 8. Из гистограммы видно, что больше всего наборов шагов в датасете соответствуют 100-процентному поркрытию и используются для подбора гиперпараметров, но в то же время имеется достаточно много методов, где такую последовательность найти пока не удалось.

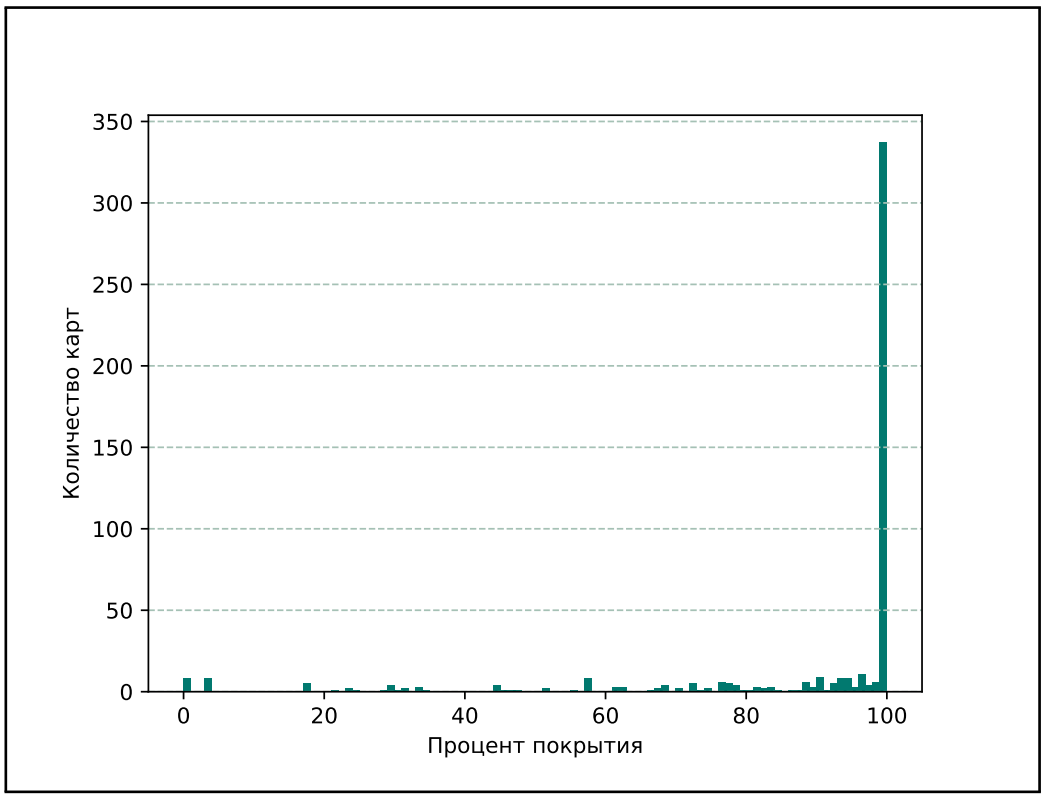


Рис. 8: Проценты покрытия в наборе данных

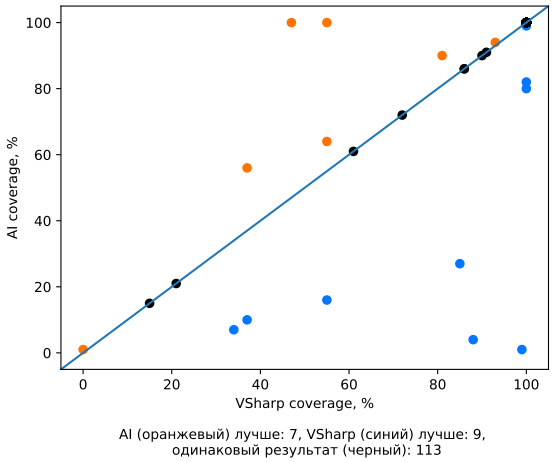
8. Результаты

На рисунке 9 представлены результаты сравнения полученной нейронной сети с алгоритмом, уже имеющимся в V# и показывающим лучшие результаты относительно других внедренных стратегий. Он называется `ExecutionTreeContributedCoverage`. На всех четырех картинках оранжевым отмечены точки, в случаях, когда обученная модель показала лучший результат, а синим в противоположных ситуациях. Черный цвет соответствует одинаковым значениям.

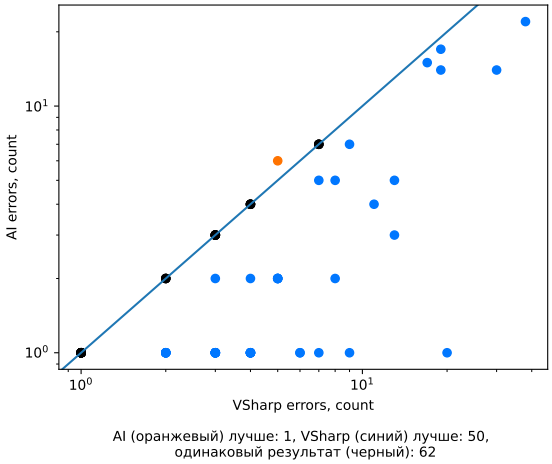
Сравнение количества найденных ошибок, сгенерированных тестов и затраченного времени производилось на методах, где было получено одинаковое покрытие. Из-за дискретности величин некоторые точки могли наложиться друг на друга из-за чего значений на графике может оказаться меньше, чем заявлено в описании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

Результаты показали, что на данный момент при сравнимых результатах по проценту покрытия и затраченному времени основным преимуществом нового подхода является генерация значительно меньшего

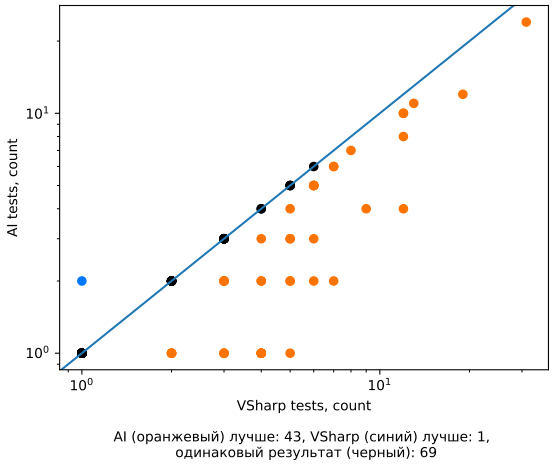
числа тестов. При этом нейронная сеть находит значительно меньше ошибок. Это можно объяснить тем, что при обучении кортежи результатов сравнивались в лексикографическом порядке, при этом на первых двух позициях стояли процент покрытия и число тестов, а число ошибок находилось в конце. Соответственно, изменение первых двух значений сильнее влияло на изменение набора данных.



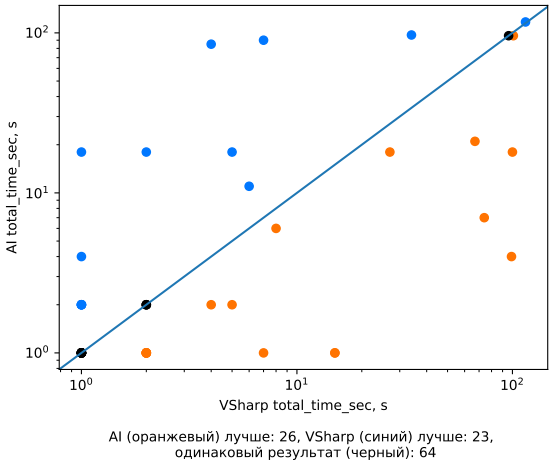
(a) Сравнение стратегий по полученному проценту покрытия



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



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



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

Рис. 9: Результаты сравнения обученной модели с лучшей стратегией, внедренной в V#

Заключение

В данной работе было проведено исследование применимости графовых нейронных сетей к нахождению путей при символьном исполнении. Были получены следующие результаты.

- Разработана инфраструктура для обучения с учителем графовой нейронной сети.
- Проведены эксперименты, результаты которых оказались сравнимыми с существующими решениями по проценту покрытия и времени работы, а по числу сгенерированных тестов предложенный подход оказался лучше.

Таким образом, графовые нейронные сети применимы к выбору шагов во время символьного исполнения программы.

Исходный код доступен по ссылке[7]. Разработка инфраструктуры, описанной в работе, велась от пользователя `Anya497`.

Развитие исследования

Возможны следующие варианты развития исследования:

- Проведение серии экспериментов, нацеленной на увеличение числа найденных ошибок;
- Подбор гиперпараметров с не фиксированным числом эпох;
- Обучение модели с использованием других символьных машин.

[7] `https://github.com/gsvgit/PySymGym/tree/dev` — ссылка на репозиторий проекта (дата доступа: 29 мая 2024 г.).

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] 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.

[8] 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.

[9] Topology adaptive graph convolutional networks / Jian Du, Shanghang Zhang, Guanhang Wu et al. // CoRR. — 2017. — Vol. abs/1710.10370. — arXiv : 1710.10370.

[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] 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.