Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.М07-мм

Разработка набора инструментов для обучения искусственных нейронных сетей выбору оптимального пути для символьного исполнения

Нигматулин Максим Владиславович

Отчёт по учебной практике
в форме «Производственное задание»

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

Санкт-Петербург
2023

Оглавление

Введение

Тестирование программного обеспечения — обязательная часть промышленных проектов и критической программно-аппаратной инфраструктуры и распространенный метод проверки качества ПО. Однако в большинстве случаев написание тестов — труд, выполняемый командой разработчиков. Сокращение затрат на проверку ПО может быть достигнуто путем автоматической генерации тестов. Одна из техник, которые могут быть использованы для автоматизации генерации тестов программного обеспечения — символьное исполнение [8]. Последнее десятилетие в мире активно ведется внедрение информационных технологий в критически важные отрасли, например, в энерго- и космическую промышленность, медицину [source], и естественно, что такие отрасли требуют серьезного подхода к обеспечению корректности используемых программных продуктов. Один из способов проверки корректности программного кода — тестирование: дополнительная программная (программно-аппаратная) стуктура, позволяющая проверить функциональность на ожидаемое поведение. Однако, как утверждал Дейкстра, «Тестирование программы может весьма эфективно продемонстрировать наличие ошибок, но безнадежно неадекватно для демонстрации их отсутствия»[source]. Соответственно для проверки критического кода необходим другой, более емкий и надежный способ.

Одним из наиболее надежных способов проверки корректности программ является символьное исполнение. Символьное исполнение — это техника анализа ПО, позволяющая выполнить программу не с конкретными входами, а с их символьным представлением. Такой подход позволяет с помощью эффективных решателей логический уравнений перебрать все возможные состояния исполнения, в которые может попасть программа, таким образом явно проверяя каждое из них на соответствие условию корректности.

Являясь методом перебора, символьное исполнение ограничено пространством поиска, что влечет за собой проблему: экспоненциальное увеличение количества («взрыв») путей, которые нужно исследовать,

чтобы зайти в целевые состояния. Сообщество активно работает над решением этой проблемы: с 2006 года проводятся соревнование решателей логики SMT-COMP[1], появляются новые движители символьного исполнения, например, V♯, ведется работа над улучшением уже существующих [klee source]

В последние годы были опубликованы несколько статей, использующих нейронные сети для более направленного поиска целевых состояний [sources]. И хотя результаты работ в этой области выглядят многообещающе, все еще существуют идеи с большим потенциалом для разработки, например, применение графовых нейронных сетей в качестве предсказателей для выбора состояний. Такой подход, в отличие от существующих методов, позволит учитывать информацию не только об особенностях состояний, но и о связях между ними.

Целью проекта является реализация фреймворка, осуществляющего обучение нейронных сетей решению задачи выбора путей основываясь на структуре графа программы.

[1] https://smt-comp.github.io/2023/

1. Термины

1.1. Графы

Гетерогенный граф — особый вид информационной сети, который может содержать несколько типов объектов и связей [7].

Отношение доминирования на графе определяется следующим образом: вершина v доминирует над вершиной $w \neq v$ в графе G, если любой путь из r в w содержит v [11].

1.2. Символьное исполнение и анализ программ

Поток управления программы. В императивном программировании предполагается, что процесс выполнения программы заключается в выполнении её инструкций вычислителем. В момент выполнения инструкции говорят, что она управляет вычислителем, переход к выполнению следующей называется передача управления или просто переход. Последовательность передач управления в процессе выполнения программы формирует её поток управления [ссылка на вики].

Базовый блок. Последовательность инструкций образует базовый блок, если:

1. Инструкция в каждой позиции доминирует (всегда выполняется раньше) всех инструкций в более поздних позициях.

2. Никакая другая инструкция не выполняется между двумя инструкциями в последовательности.

В таких условиях, когда выполняется первая инструкция в базовом блоке, остальные инструкции обязательно выполняются ровно один раз и по порядку. [18]

Граф потока управления (Control Flow Graph, CFG). CFG является направленным графом, имеет вершину для каждого базового блока и ребро для каждой возможной передачи управления [1].

Направленный поиск — техника поиска определенной инструкции в графе исполнения (в отличие от ненаправленного поиска, который ведется до выполнения какого-либо условия).

Статический анализ кода — анализ кода, производимый без исполнения программы (в отличие от динамического анализа).

Тестовое покрытие — отношение покрытых тестами инструкций или ветвей программы к общему количеству инструкций или ветвей, в процентах.

Зона покрытия — множество инструкций в методе, которые требуется покрыть тестами.

1.3. Машинное обучение

Эвристический алгоритм (эвристика) — алгоритм решения задачи, не являющийся гарантированно точным или оптимальным, но достаточный для решения поставленной задачи. Позволяет ускорить решение задачи в тех случаях, когда точное решение не может быть найдено.

Генетической алгоритм — это эвристический алгоритм поиска, используемый для решения задач путём случайного подбора, комбинирования и изменения параметров с использованием механизмов, аналогичных естественному отбору в природе. В процессе работы генетического алгоритма создаётся множество случайных агентов начальной популяции, каждый из которых оценивается **fitness-функцией** (функцией приспособленности) и получает свое значение **fitness** (приспособленности). При достаточной приспособленности особи вероятностно выбираются из прежней популяции, и модифицируются, чтобы начать новый цикл (или поколение). Новое поколение особей-кандидатов будет использовано в следующем поколении процесса. Алгоритм обычно завершается, когда достигнуто либо максимальное число поколений, либо достигнут удовлетворительный результат. Таким образом, каждое последующее поколение становится более подходящим для окружающей среды популяции [4].

Q-learning — это алгоритм обучения с подкреплением, позволяющий определять предпочтительность действия из набора возможных действий для каждого состояния системы. Информация о предпочитаемых действиях в определенных состояниях хранится в **Q-table**, которая обновляется по ходу обучения: каждый шаг алгоритма должен быть отмечен наградой, которая изменяет значение предпочтительности для совершаемого шага [16].

2. Обзор

Существует несколько решений, использующих машинное обучение для создания алгоритма выбора путей.

2.1. Q-KLEE

В данном подходе авторы используют Q-learning для контроля символьного исполнения. Первым шагом авторы получают доминаторы графа исполнения по отношению к определенным инструкциям с помощью статического анализа. Позитивная награда в процессе обучения возвращается только в том случае, когда символьный исполнитель вошел в вершину-доминатор, иначе возвращается негативная награда. Первичные эксперименты авторов показывают, что для достижения целевой инструкуции требуется в среднем на 90% меньше шагов по сравнению с базовой стратегией в KLEE [17].

Стоит отметить, что при заявленной эффективности подход не может быть использован в качестве алгоритма выбора путей без модификаций, так как результирующий алгоритм стремится к определенным инструкциям, а не исследует граф исполнения.

2.2. LEARCH

Другой известный подход с использованием машинного обучения — LEARCH. Авторы утверждают, что LEARCH способен выбирать перспективные состояния символьного исполнения для решения проблемы взрыва путей. LEARCH оценивает вклад каждого состояния в максимизацию покрытия в выданный квант времени, в отличие от использования эвристик, созданных вручную и основанных на простых статистиках, грубо аппроксимирующих целевой алгоритм [10]. Однако данный подход не учитывает информацию о структуре графа исполнения.

2.3. SyML

Данный подход позволяет отслеживать пути исполнения уязвимых программ и извлекать соответствующие признаки: обращения к регистрам и памяти, сложность функций, системные вызовы для поиска путей. Авторы обучают модели для изучения паттернов уязвимых путей на основе извлеченных признаков и используют их предсказания для обнаружения уязвимых путей исполнения в новых программах [14].

Однако данный подход платформозависим и не переносим, так как в разных языках могут быть разные уязвимости.

2.4. Automatic Heuristic Learning

В данном подходе авторы представляют решение для автоматической генераци эвристик для символьного исполнения. Ручная разработка хорошей поисковой эвристики нетривиальна и обычно приводит к неоптимальным и нестабильным результатам. Цель данной работы — преодоление этого недостатка путем автоматического обучения эвристикам поиска. Авторы представляют алгоритм, который эффективно находит оптимальную эвристику для каждой исследуемой программы [2].

Тем не менее, отстутствие оптимальной эвристики на редкий случай, время на ее подбор могут значительно сказаться на эффективности подхода.

Выводы

Таким образом, существующие подходы решают проблему нахождения оптмимального пути в графе исполнения для достижения целевых состояний разными способами. Данная работа представляет еще один возможный подход к оптимизации поиска пути.

3. Постановка задачи

Цель работы: реализовать фреймворк, выполняющий генерацию моделей-учителей в ходе обучения с помощью символьной машины V♯ [19].

Поставленные задачи:

1. Создать протокол общения с сервером обучения для получения сигнала об окончании взаимодействия, информации о награде за шаг и состоянии символьного исполнения во время обучения;
2. Создать фреймворк, использующий генетическое обучение для создания и обучения нейронных сетей во время взаимодействия с сервером обучения как с игровой средой;
3. Поддержать возможность одновременного обучения нескольких нейронных сетей;
4. Поддержать возможность использования GPU для ускорения работы нейронных сетей.

4. Описание предлагаемого решения

В процессе работы механизм символьного исполнения, или символьная машина, на каждой итерации продвигает одно из доступных состояний по графу исполнения. При этом конечная цель — сгенерировать тесты, обеспечивающие максимальное тестовое покрытие, сделав при этом минимальное количество таких итераций. Выбор состояния на каждой итерации управляется эвристиками, так как заранее знать, какое из них приведёт к успеху, невозможно [13]. В качестве функции выбора состояния исполнения для продвижения, предлагается использовать графовую нейронную сеть, которая по графу, описывающему текущий исследованый граф исполнения, будет предсказывать, какое состояние исполнения сейчас выгоднее всего выбрать.

Этот процесс можно представить как партию в настольную игру, или просто «игру», где игроком является нейронная сеть, картой игрового мира — CFG программы, фишками — состояния исполнения. Каждый ход игрок выбирает фишку, таким образом предсказывая перспективное состояние исполнения. Далее символьная машина двигает ее, таким образом меняя состояние исполнения. После предсказания игрок-нейронная сеть получет за свое действие награду. Задача игрока — за наименьшее количество ходов обойти как можно больше целевых состояний на игровой карте.

4.1. Модель графовой нейронной сети

Для извлечения отношений между узлами графа предлагается использовать графовые нейронные сети [5]. Состояние символьной машины можно описать как множества блоков инструкций-вершин, посещенных или доступных для посещения, и множества состояний-фишек, которые двигаются по блокам-вершинам.

На рисунке 1 изображен вход нейронной сети: гетерогенный граф, содержащий структуру графа исполнения и состояний исполнителя. Для каждой вершины сохраняются следующие аттрибуты:

1. VisitedByState — вершина посещена состоянием и состояние вышло из нее;

2. TouchedByState — вершина посещена состоянием, но состояние из нее не вышло;

3. InCoverageZone — вершина в зоне покрытия;

Для каждой вершины с состоянием граф хранит следующие аттрибуты:

1. PathConditionSize — размер условия пути, считается как размер логического выражения;

2. VisitedAgainVertices — количество повторно посещенных вершин;

3. VisitedNotCoveredVerticesInZone — количество посещенных и не покрытых вершин в зоне покрытия;

4. VisitedNotCoveredVerticesOutOfZone — количество посещенных и не покрытых блоков вне зоны покрытия;

Во входном графе существует четыре типа ребер:

1. Ребенок состояния — ориентированное ребро, указывающее на состояния, порожденные текущим;

2. A находится в B — существует между состоянием и вершиной, если состояние A находится в вершине B;

3. Наследник вершины — ориентированное ребро, указывающее на блоки инструкций, которые нужно исполнить после текущего:

4. Ребро истории — ребро, содержащее количество посещений блока данным состоянием.

Для каждого состояния нейронная сеть предсказывает вектор признаков:

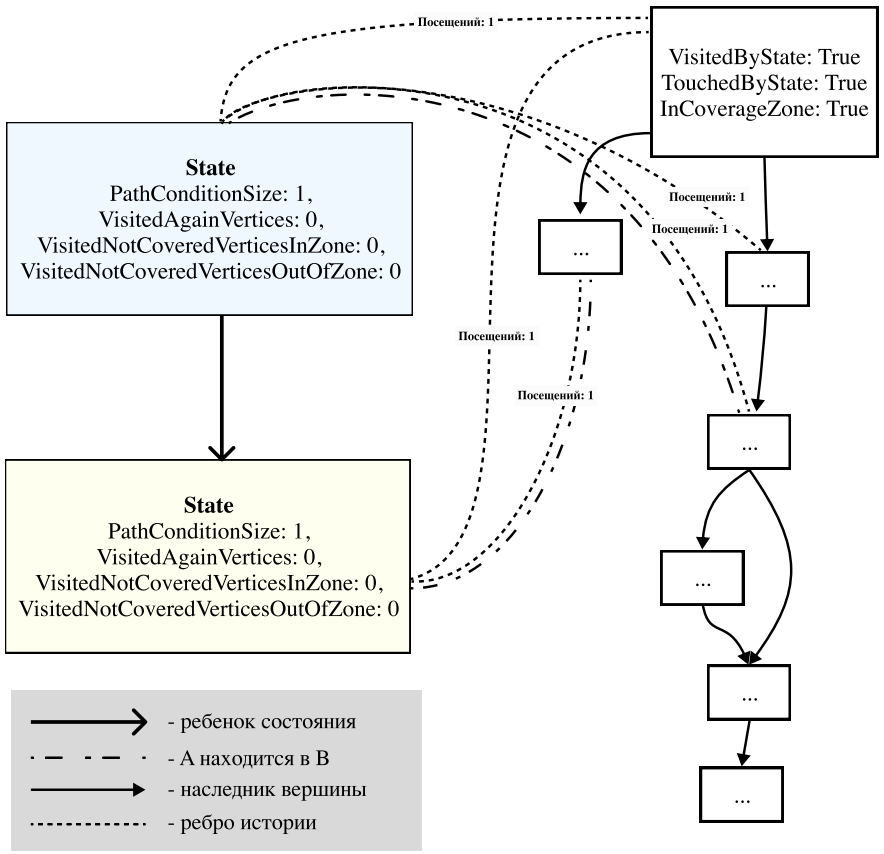

Рис. 1: Входной граф

1. NextInstructionIsUncoveredInZone — следующая инструкция находится в зоне покрытия, но не покрыта;

2. ChildNumberNormalized — количество состояний, когда-либо порожденных текущим, нормализованное к максимальному;

3. VisitedVerticesInZoneNormalized — количество посещенных инструкций в зоне покрытия, нормализованное к максимальному;

4. Productivity — отношение всех посещенных инструкций к количеству повторных посещений;

5. DistanceToReturnNormalized — расстояние до ближайшего return, нормализованное к максимальному;

6. DistanceToUncoveredNormalized — расстояние до ближайшего непокрытого блока, нормализованное к максимальному;

7. DistanceToNotVisitedNormalized — расстояние до ближайшего непосещенного блока, нормализованное к максимальному;

8. ExpectedWeight — референсное значение веса состояния.

4.2. Модули обучения

На рисунке 2 схематично приведен процесс обучения нейронной сети выбору перспективных состояний, состоящих из трех этапов: предобучения, генетического обучения, постобучения. Далее приводится описание каждого из этапов.

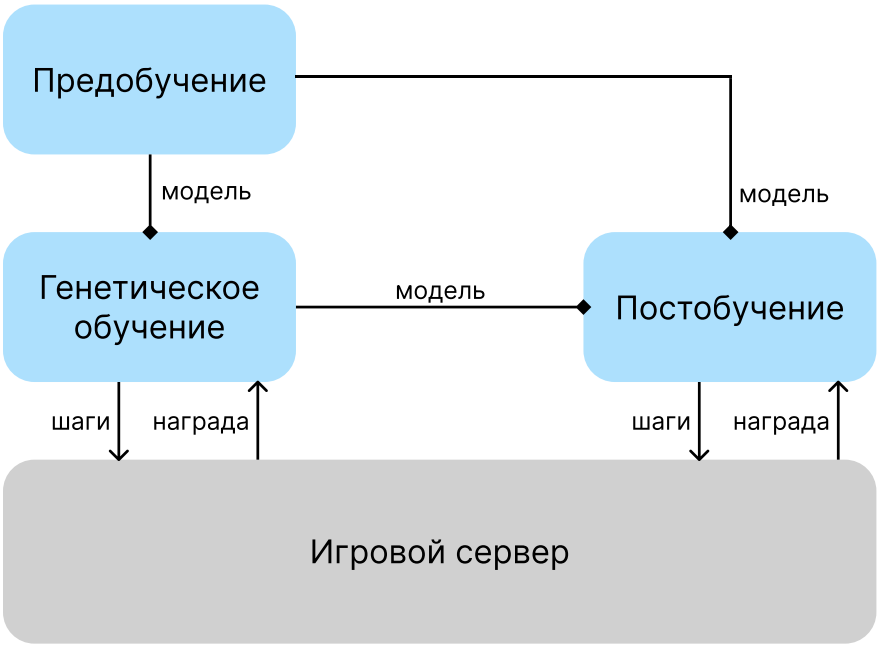

Рис. 2: Структура модулей обучения

4.2.1. Предобучение

На данном этапе происходит предобучение нейронной сети предсказанию признаков состояния исполнения из датасета по входному гетерогенному графу. Датасет состоит из графов исполнения и эталонных вектора значений для каждого состояния в этом графе. В качестве оптимизатора используется Adam [9]. В качестве функции потерь ис-

пользуется средняя квадратичная ошибка[2]. Потери для каждого графа вычисляются как усредненное значение ошибки по всем состояниям в графе.

4.2.2. Генетическое обучение

В качестве эталонных данных для постобучения нейронной сети, используются последовательности выбора состояний, или пути, по которым должна двигаться символьная машина и которые считаются оптимальными[3]. Чтобы создать набор нейронных сетей, способных генерировать такие пути, используется метод стохастического программирования: множество нейронных сетей с различными весами в процессе генетического обучения генерируют различные пути для различных программ, в процессе смешивая веса веса и соревнуюясь между собой.

В качестве исходных данных для датасета используются исполняемые файлы с различными алгоритмами с различными по сложности для символьной навигации графами исполнения. Неполный листинг игровых карт-методов можно найти в конфигурационных файлах [15].

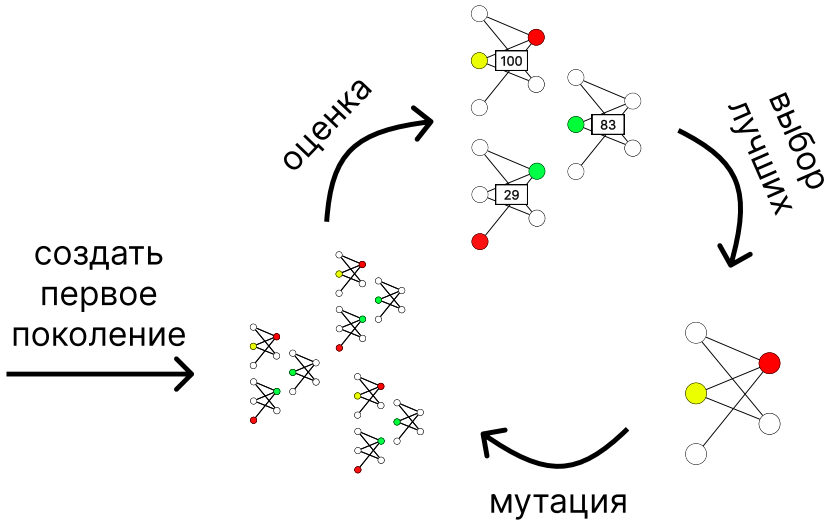

Рис. 3: Структура генетического обучения

[2] https://pytorch.org/docs/stable/generated/torch.nn.functional.mse_loss.html
[3] алгоритма для подбора оптимальных не существует [13]

На рисунке 3 изображена структура генетического обучения, реализованная во фреймворке. Сперва фреймворк генерирует нейронные сети с заданной архитектурой и случайными весами. Такие нейронные сети добавляются в общий пул вместе с предобученными для формирования первого поколения.

В процессе обучения для каждой игры агента вычисляется кортеж

$$\langle coverage, steps, tests, errors \rangle, \tag{1}$$

где coverage — покрытие в процентах, steps — количество шагов, сделанных агентом на карте, tests — количество сгенерированных тестов, errors — количество найденных ошибок.

Для вычисления fitness-функции агента обучения для каждого вектора из элементов кортежа с одинаковым индексом рассчитывается евклидово расстояние до идеального вектора. Таким образом вычисляется кортеж расстояний до идельных векторов, который используется как fitness в генетическом обучении.

Далее для подачи на вход генетического алгоритма нейронная сеть отображается в одномерный вектор, состоящий из весов ее слоев. Для формирования следующего поколения используются значения fitness, полученные на шаге оценки. В соответствии с этими значениями веса нейронных сетей будут или не будут использованы в качестве родителей.

Каждой нейронной сети соответствует значение fitness, в согласии с которым веса модели будут или не будут использованы для формирования следующего поколения. Далее вектор отображается обратно в нейронную сеть, значение fitness которой вновь оценивается во время игры с сервером.

4.2.3. Постобучение

Последняя стадия обучения состоит из трех частей: генерация датасета, обучение на полученных данных с помощью градиентного спуска

и валидация. Данный этап использует нейронные сети сгенерированные во время работы генетического обучения.

В качестве датасета используются те же данные, что и во время генетического обучения. Каждую эпоху обучения для обучаемой модели после игры на карте сохраняется кортеж, аналогичный структуре fitness генетического обучения 4.2.2, который отражает статистику совершенных шагов на текущем пути. Если на текущей карте результаты превосходят предыдущий эталон, то старая последовательность шагов заменяется на новую и результаты на карте обновляются.

5. Реализация

В данном разделе представлено описание деталей реализации разрабатываемого фреймворка для генетического обучения на языке PYTHON3. Исходный код приложен по ссылке в GITHUB[4].

5.1. Архитектура фреймворка

Фреймворк состоит из нескольких компонентов, отношения между которыми изображены на рисунке 4:

- Сервер обучения — служит в качестве игрового окружения;
- Класс Connector — обуспечивает соединение с сервером;
- Функция play_game — осуществляет взаимодействие с сервером через Connector;
- Класс Broker — хранит результаты обучения, управляет соединениями с сервером;
- Genetic Learning Framework — использует play_game для оценивания агентов обучения;
- Statistics Module — сохраняет и отображает статистику обучения.

Протокол общения. Для реализации протокола общения с серверами обучения были выбраны сокеты. Логику общения с сервером инкапсулирует класс Connector. Connector во время общения с сервером может получить и обработать сообщения, содержащее актуальное состояние графа исполнения, информацию о награде за шаг, предсказанный агентом-игроком, наименования карт, которые может сыграть агент, информацию о том, что игра окончена. Также Connector в процессе работы отправляет серверу сообщения о начале игры, о сделанном шаге, сообщение с запросом карт, сообщение для переключения режима работы на тестовую и валидационную выборки.

[4] https://github.com/VSharp-team/VSharp

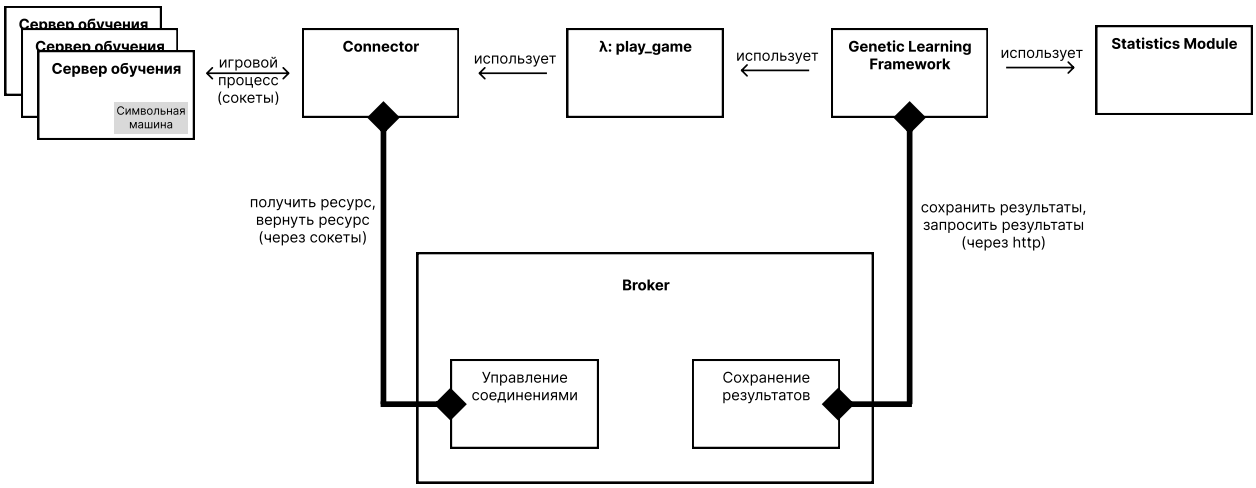

Рис. 4: Архитектура фреймворка

Одновременное обучение. Для поддержки одновременного обучения нескольких агентов был реализован брокер сокетов: когда очередной процесс готов осуществлять обучение, он соединяется с асинхронным брокером по http, запрашивает у него ресурс — сокет запущенного сервера обучения. По окончании обучения процесс возвращает ресурс, брокер уничтожает сервер.

Статистика, логгирование, таблицы с результатами обучения. Результаты игры каждого агента отправляются брокеру, в конце каждого поколения фреймворк обрабатывает, ранжирует и записывает результаты игр в таблицу. Таже для оценки времени работы нейронной сети фрейморк собирает статистику работы всех нейронных сетей в поколении. Логи работы брокера, серверов разделены по уровням, записываются в общий файл.

5.2. Использованные технологии

Для реализации генетического обучения был выбрана библиотека PyGAD [3]: этот инструмент реализует базовые стратегии генетического обучения, позволяет определять свои, поддерживает ускорение обучение через использование нескольких потоков.

Обучение осуществлялось с использованием символьной машины V♯. Для этого был написан сервер, осуществляющий операции с символь-

ной машиной и принимающий команды и высылающий информацию с помощью сокетов на клиент обучения.

Для представления в памяти и обучения нейронных сетей использовалась библиотека PYTORCH [12]. PYTORCH создана для работы с нейронными сетями, содержит множество высокоуровневых утилит для обучения, импорта и экспорта нейронных сетей, позволяющих ускорить разработку.

Ускорение обучения. Для реализации параллельного обучения были использованы возможности PyGAD: внутри главной функции существует пул процессов, который используется для параллельной оценки fitness-функции нескольких агентов. Для переноса данных на GPU используются встроенные инструменты PYTORCH.

6. Эксперимент

Проверим применимость генетического обучения и графовых нейронных сетей (GNN) для решения задачи выбора путей.

6.1. Условия эксперимента

Для тестирования на платформе JVM были выбраны первые 200 тестов из проекта GUAVA [6] версии 28.0.1. Сравнение осуществлялось по тем 117 тестам, на которых завершились все 4 алгоритма.

6.2. Исследовательские вопросы

RQ1: Сравнимы ли результаты работы алгоритма с существующими подходами?

6.3. Метрики

RQ1: Сравнение с существующими алгоритмами будем производить по ряду параметров: среднее покрытие (больше-лучше), количество сделанных шагов для методов, покрытых на 100% (меньше-лучше), количество сгенерированных тестов для методов, покрытых на 100% (меньше-лучше), количество найденных ошибок (больше-лучше).

6.4. Результаты

В таблицах 1, 2, 3, 4 представлены результаты работы и сравнение результатов существующих эвристик с разработанным алгоритмом.

RQ1: Можно заметить, что хоть среднее покрытие разработанного алгоритма незначительно уступает некоторым методам (1), но при этом до 100% модель доходит в среднем за наименьшее количество шагов (2)и генерирует наименьшее количество тестов (3), в среднем генерирует меньше ошибок (4). Это может говорить о более узконаправленном поиске инструкций.

Таблица 1: Среднее покрытие

стратегия	E	медиана	δ
BFS	**80.87**	**100.0**	**32.43**
FORK_DEPTH_RANDOM	80.65	100.0	32.39
GNN	80.65	100.0	32.88
FORK_DEPTH	**80.87**	**100.0**	**32.43**

Таблица 2: Шаги для методов, покрытых на 100%

стратегия	E	медиана	δ
BFS	198.46	25.5	549.21
FORK_DEPTH_RANDOM	122.33	25.0	302.03
GNN	**73.47**	**25.5**	**136.6**
FORK_DEPTH	183.74	25.5	651.32

Таблица 3: Сгенерированные тестов для методов, покрытых на 100%

стратегия	E	медиана	δ
BFS	4.29	1.0	9.9
FORK_DEPTH_RANDOM	3.05	1.0	5.48
GNN	**1.96**	**1.0**	**2.53**
FORK_DEPTH	2.4	1.0	3.19

Таблица 4: Ошибки для методов, покрытых на 100%

стратегия	E	медиана	δ
BFS	**0.9**	**0**	**3.63**
FORK_DEPTH_RANDOM	0.85	0	4.06
GNN	0.52	0	2.31
FORK_DEPTH	2.2	0	13.34

7. Применение

Полученный алгоритм был запущен на символьной машине для JVM, показал результаты, сравнимые со встроенными алгоритмами и лучше. Данная работа является фундаментом для продолжения исследования применимости графовых нейронных сетей для задачи выбора путей исполнения.

Заключение

В результаты работы были выполнены следующие задачи:

1. Создан протокол общения с сервером обучения для получения сигнала об окончании взаимодействия, информации о награде за шаг и состоянии символьного исполнения во время обучения;
2. Создан фреймворк, использующий генетическое обучение для создания и обучения нейронных сетей во время взаимодействия с сервером обучения как с игровой средой;
3. Поддержана возможность одновременного обучения нескольких нейронных сетей;
4. Поддержана возможность использования GPU для ускорения работы нейронных сетей.

В будущем планируется пересмотреть архитектуру проекта, написать веб-сервис для просмотра результатов обучения, внедрить CI в проект.

Список литературы

[1] Cooper Keith D., Torczon Linda. Chapter 4 - Intermediate Representations // Engineering a Compiler (Third Edition) / Ed. by Keith D. Cooper, Linda Torczon. — Philadelphia : Morgan Kaufmann, 2023. — P. 159–207. — URL: https://www.sciencedirect.com/science/article/pii/B9780128154120000103.

[2] 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.

[3] Gad Ahmed Fawzy. PyGAD: An Intuitive Genetic Algorithm Python Library // CoRR. — 2021. — Vol. abs/2106.06158. — arXiv : 2106.06158.

[4] Alam Tanweer, Qamar Shamimul, Dixit Amit, Benaida Mohamed. Genetic Algorithm: Reviews, Implementations, and Applications. — 2020. — 2007.12673.

[5] The Graph Neural Network Model / Franco Scarselli, Marco Gori, Ah Chung Tsoi et al. // IEEE Transactions on Neural Networks. — 2009. — Vol. 20, no. 1. — P. 61–80.

[6] Guava. — https://guava.dev. — Accessed: 2023-09-23.

[7] Heterogeneous Graph Attention Network / Xiao Wang, Houye Ji, Chuan Shi et al. // CoRR. — 2019. — Vol. abs/1903.07293. — arXiv : 1903.07293.

[8] Kapus Timotej, Cadar Cristian. Automatic testing of symbolic execution engines via program generation and differential testing // 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). — 2017. — P. 590–600.

[9] Kingma Diederik P., Ba Jimmy. Adam: A Method for Stochastic Optimization. — 2017. — 1412.6980.

[10] 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. — CCS '21. — New York, NY, USA : Association for Computing Machinery, 2021. — P. 2526–2540. — URL: https://doi.org/10.1145/3460120.3484813.

[11] Lengauer Thomas, Tarjan Robert Endre. A Fast Algorithm for Finding Dominators in a Flowgraph // ACM Trans. Program. Lang. Syst. — 1979. — jan. — Vol. 1, no. 1. — P. 121–141. — URL: https://doi.org/10.1145/357062.357071.

[12] PyTorch: An Imperative Style, High-Performance Deep Learning Library / Adam Paszke, Sam Gross, Francisco Massa et al. // Advances in Neural Information Processing Systems 32. — Curran Associates, Inc., 2019. — P. 8024–8035. — URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-pdf.

[13] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Comput. Surv. — 2018. — may. — Vol. 51, no. 3. — 39 p. — URL: https://doi.org/10.1145/3182657.

[14] 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.

[15] VSharp game maps listing. — https://github.com/emnigma/vsharp_benchmarks/blob/3c23897b778e9db840680dc160fadd5e9f28cd5f/prebuilt/game_maps.json. — Accessed: 2023-09-23.

[16] Watkins Christopher J. C. H., Dayan Peter. Q-learning // Machine Learning. — 1992. — . — Vol. 8, no. 3. — P. 279–292. — URL: https://doi.org/10.1007/BF00992698.

[17] Wu Jie, Zhang Chengyu, Pu Geguang. Reinforcement Learning Guided Symbolic Execution // 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). — 2020. — P. 662–663.

[18] Yousefi Javad, Sedaghat Yasser, Rezaee Mohammadreza. Masking wrong-successor Control Flow Errors employing data redundancy // 2015 5th International Conference on Computer and Knowledge Engineering (ICCKE). — 2015. — P. 201–205.

[19] V♯. — https://github.com/VSharp-team/VSharp. — Accessed: 2023-09-23.