VKA miqyosini tahlil qilishning yaxshiroq usullari orqali fizik, kimyo va matematika sohalarida ilg’or natijalarga erishish kutilmoqda. Bular, ehtimol, gradient miqyosi va boshqa masshtablash aspektlarini, masalan, mahalliy minimallarning zichligi va xarajat landshaftining shaklini o‘z ichiga oladi. Ushbu fundamental natijalar kvant ustunligini izlashga yordam beradi.

Foydalanilgan adabiyotlar ro‘yxati:

1. Zhao, A. et al. Measurement reduction in variational quantum algorithms. Phys. Rev. A 101, 062322 (2020).
2. Beckey, J. L., Cerezo, M., Sone, A. & Coles, P. J. Variational quantum algorithm for estimating the quantum Fisher information. Preprint at https://arxiv.org/abs/2010.10488 (2020).
3. Arrasmith, A., Cincio, L., Somma, R. D. & Coles, P. J. Operator sampling for shot-frugal optimization in variational algorithms. Preprint at https://arxiv.org/ abs/2004.06252 (2020).
4. Scherer, A. et al. Concrete resource analysis of the quantum linear-system algorithm used to compute the electromagnetic scattering cross section of a 2D target. Quantum Inf. Process. 16, 60 (2017).
5. Magann, A. B. et al. From pulses to circuits and back again: a quantum optimal control perspective on variational quantum algorithms. Phys. Rev. X Quantum 2, 010101 (2021).
6. Santha, M. Quantum walk based search algorithms. in Theory Appl. Model. Comput. 4978, 31–46 (2008).
7. Parrish, R. M., Iosue, J. T., Ozaeta, A. & McMahon, P. L. A Jacobi diagonalization and Anderson acceleration algorithm for variational quantum algorithm parameter optimization. Preprint at https://arxiv.org/abs/ 1904.03206 (2019).

УЛУЧШЕНИЕ ОПТИМИЗАЦИИ ПАТЧА ДЛЯ МНОГОСЕКЦИОННЫХ ОШИБОК ПРИ АВТОМАТИЗИРОВАННОМ ВОССТАНОВЛЕНИИ ПРОГРАММ

[1]Abdinabiev Aslan Safarovich,
[2]Turdiyev Sirojiddin Sayfiddin o‘g‘li
[1]Dept. of Computer Science, University of Seoul, South Korea
[2]Toshkent Davlat Transport Universiteti, O‘zbekiston
sayfiddinovichsirojiddin@gmail.com

Аннотация: Методы автоматизированного исправления программ, основанные на глубоком обучении, продемонстрировали значительные улучшения в устранении ошибок. В этих методах часто используются предварительно обученные модели нейро-машинного перевода для создания исправлений для ошибочного исходного кода. Для исходного кода с множеством ошибок в разных местах были предложены различные подходы, в том числе генерация множества патчей-кандидатов для каждого дефектного участка и их

объединение для формирования окончательных патчей. Однако задача состоит в том, чтобы ранжировать эти сгенерированные патчи по вероятности их правильности и выбрать конкретное число из верхней части, так как объединять их все нецелесообразно. В данной работе мы представляем усовершенствованную методику оптимизации патчей, которая решает эту задачу в три этапа: отсеивание ненужных патчей, ранжирование и отбор наиболее ранжированных патчей, а также их объединение. В экспериментах с использованием набора данных данный метод показал лучшую производительность по сравнению с другими исследованиями.

Введение: Автоматизированный исправления программ (APR) предназначен для автоматического исправления программных ошибок без вмешательства человека. Автоматизированный восстановительный программный процесс (APR) предназначен для автоматического исправления ошибок в программном обеспечении без вмешательства человека. Распространенной проблемой в APR является устранение многосекционных ошибок, которые включают в себя несколько наборов непрерывных строк исходного кода, связанных с одной ошибкой. Одним из способов исправления многосекционных ошибок является генерация множества патчей-кандидатов с помощью модели нейро-машинной трансляции (NMT) и их комбинирование. Однако такой подход приводит к экспоненциальному увеличению пространства патчей в зависимости от количества комбинаций. Поэтому Дж. Ким и др. [1] предложили стратегию оптимизации патчей для эффективного отсеивания (фильтрация патчей), ранжирования (ранжирование патчей) и комбинирования (комбинирование патчей) патчей-кандидатов, сгенерированных его методом багги-блоков и тонкой настройкой CodeBERT. Однако, по нашим наблюдениям, этот оптимизационный подход имеет ряд ограничений, особенно на этапе ранжирования и комбинирования патчей. Поэтому в данной работе предлагается усовершенствованная методика оптимизации патчей для многозвенных ошибок. Кроме того, мы уделяем особое внимание шагу исправления программы для устранения выявленных ошибок и стилю generate and validate (G&V) для применения сгенерированного патча и его оценки на тестовых примерах. Обобщая вышесказанное, можно сказать, что мы вносим следующий вклад:

- Предлагаем усовершенствованную оптимизацию патчей для более эффективного исправления многосекционных ошибок.
- Анализируем проблемы предыдущей оптимизации патчей и предлагаем их решения. Проводим эксперименты с нашими решениями и представляем результаты.
- Мы фокусируемся на многосекционных ошибках, которые являются частыми и сложными проблемами при разработке программного обеспечения, чтобы повысить надежность программного обеспечения и производительность разработчиков.

Сначала мы генерируем initialtopKs - начальную версию topKs, используя следующее уравнение (уравнение 4).

$$k_i = max\ (\{k \mid k^{cn} \leq \frac{M \times bLocs_i{}^{cn-1}}{\prod_{j=1}^{cn} bLocs_{j,\ j \neq i}},\ k \leq SP\}) \qquad (4)$$

Это уравнение генерирует значение k_i для каждого $cd(i)$, удовлетворяющего первому условию. Как видно из рис.3, значения $initialtopKs$, генерируемые с помощью этого уравнения, изменяются пропорционально значениям $bLocs$. Однако если посчитать количество комбинированных патчей, которые мы можем создать с помощью этих начальных $initialtopKs$, то получим 5000, что слишком мало и не удовлетворяет второму условию. Решить эту проблему можно, увеличив некоторые элементы в $initialtopKs$. Поскольку вариантов таких приращений много, то для поиска наиболее оптимального из них воспользуемся алгоритмом 1.

Algorithm 1: Get optimal $topKs$

```
    INPUT:
    M, multiplier, initialtopKs, SP
1   function getOptimalTopKs(M, SP, initialtopKs, multiplier):
2       mainList ← []
3       for each k from initialtopKs do:
4           product ← (∏_{j=1}^{kl} initialtopKs(j)) / k   // calculate a product of the
5   other items
6           kIncrementsList ← []
            iterator ← k
7           while True do:
8               add iterator to kIncrementsList
9               if iterator * product ≤ M then
10                  if iterator + 1 ≤ min(k * mutiplier, ceil(SP * k/
11  sum(initialtopKs))):
12                      increment iterator
13                  else
                        break
14              Else
15                  break
16          add kIncrementsList to mainList
17      combinations = makeCombinations(mainList)
18      filteredcombinations = filterCombinations(M, combinations)
19      sortedcombinations = sortCombinations(filteredcombinations)
20  return the first item of sortedcombinations
```

Алгоритм 1 принимает в качестве входных данных *initialtopKs* и некоторые параметры. Сначала формируется список *mainList* , в котором собраны списки всех возможных приращений каждого элемента *initialtopKs*. Строки 2-16 алгоритма1 описывают детали этого процесса. Мы контролируем ограничение с помощью множителя при увеличении значения каждого элемента (строка 10), чтобы предотвратить ошибку памяти при слишком большом количестве

ошибочных кодов (*if* $cn = 20$ *and all of the buggy codes have a single buggy location, the number of items in* $itemincrementsList$ *of each* k *will increase up to 10000 without this limitation and raise a memory error when combining*).

После генерации *mainList* (см. рис. 3) с помощью *mainList* (строка 17) создаем список комбинаций, в котором собраны все возможные комбинации k значений. Затем отфильтровываем все комбинации, произведение элементов которых превышает M (*such as the last combination in* рис. 3). После фильтрации у нас будут все варианты приращений, которые мы можем выполнить. Чтобы найти наиболее оптимальный для наших условий, отсортируем их по произведению их элементов на большее и стандартному отклонению элементов на меньшее (строка 19). Наконец, мы возвращаем первый элемент отсортированной комбинации в качестве оптимального

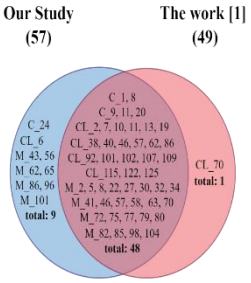
$topKs$ *(In Puc.3, the resulted* $topKs = (5, 10, 10, 4, 5)$*).*

Результаты экспериментов

Вносят ли эти улучшения вклад в повышение производительности?

Мы сравниваем наше исследование с базовым [1] по результатам, оцененным на 24, 133 и 106 ошибках из модулей *Chart, Closure* и *Math* набора данных *Defects4j[7]*.

Рис 4. Сравнение эффективности (C: Chart, CL: Closure, M: Math)



На рис. 4 представлен результат сравнения, где показаны ошибки, исправленные обеими работами. Как видно из рисунка, в результате использования улучшенной стратегии оптимизации патчей технология восстановления программы исправила 57 ошибок, что на 8 больше, чем в базовой версии (относительное улучшение на *16%*). Здесь из числа исправленных ошибок работы [1] была исключена ошибка M_18, поскольку она не проверяла границы кодировки и была некорректной. Более того, 9 ошибок были исправлены только с помощью улучшенной оптимизации патча, причем пять из них являются многокусковыми (ошибки делятся на однокусковые и многокусковые в зависимости от количества используемых для исправления кусков), включая CL_6, M_43, M_56, M_62 и M_86. Кроме того, мы успешно исправили оставшиеся четыре однокусковые ошибки. Это свидетельствует об эффективности наших улучшений, выполненных на этапах ранжирования и комбинирования. Таким образом, можно сказать, что наши улучшения способствовали повышению производительности.

В данной работе мы предложили усовершенствованную методику оптимизации патчей, генерируемых методом исправления ошибок на основе глубокого обучения, для кодов с ошибками, состоящими из нескольких блоков.

В рамках данного подхода были внесены некоторые изменения в работу [1], направленные на повышение ее производительности и применимости. Во-первых, мы обновили этап ранжирования патчей, улучшив качество ранжирования, а затем предложили новый метод комбинирования патчей для более эффективного объединения патчей. Мы провели эксперименты и показали эффективность предложенной нами работы.

Список литературы:

1. J. Kim and B. Lee, “MCRepair: Multi-Chunk Program Repair via Patch Optimization with Buggy Block”, In proc. of the 38th Annual ACM/SIGAPP Symposium on Applied Computing (SAC 2023), pp. 1508-1515, 2023.

2. N. Jiang, T. Lutellier, and L. Tan, “CURE: Code-Aware Neural Machine Translation for Automatic Program Repair,” In proc. of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE), pp. 1161-1173, 2021.

3. S. Wang et al, “Automated patch correctness assessment: How far are we?,” In proc. of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 968-980, 2020.

4. T. Lutellier, H.V. Pham, L. Pang, ..., and L. Tan, " CoCoNuT: combining context-aware neural translation models using ensemble for program repair", In proc. of the 29th ACM SIGSOFT international symposium on software testing and analysis (ISSTA), pp. 101-114, 2020.

5. Y. Li, S. Wang, and T.N. Nguyen, “DEAR: a novel deep learning-based approach for automated program repair.,” In Proc. of the 44th IEEE/ACM International Conference on Software Engineering (ICSE), pp. 511–523, 2022.

6. https://sites.google.com/view/learning-fixes/home?authuser=0

7. https://github.com/rjust/defects4j

MA’LUMOTLARNI INTELLEKTUAL TAHLILLASH UCHUN OLDINDAN TASNIFLASH VA YAKUNIY KONVOLYUTSION NEYRON TARMOG‘I

Raximov Nodir Odilovich
Toshkent axborot texnologiyalari universiteti
Shirinboyev Ravshan
O‘zbekiston Milliy universitetining Jizzax filiali
Saidova Zilola Habibovna
Toshkent Kimyo xalqaro universiteti

Annotatsiya: Ushbu maqola konvolyutsion neyron tarmoqlari va ma’lumotlar tahlili bo’yicha muhim ma’lumotlar taqdim etadi. Bu tarmoqlar ma’lumotlarni tahlil qilishda va tasniflashda foydalaniladigan texnologiyani tavsiflaydi. SNA-1(Social Network Analysis) va SNA-2 (Social Network Analysis) nomli ikkita neyron tarmoqi tuzilishi va ishlash prinsiplari haqida ma’lumot berilgan. Bu tarmoqlar ma’lumotlarni