Инженерный вестник Дона, №2 (2018)
ivdon.ru/ru/magazine/archive/n2y2018/4945

Совершенствование статического анализа программного кода на основе графа явных вызовов

А.Ю. Федоров, Е.М. Портнов

Национальный исследовательский университет «Московский институт электронной техники»

Аннотация: Целью данного исследования является создание алгоритмов статического анализа для поиска всевозможных последовательностей вызовов функций, приводящих к определенной точке в программе. Для достижения данной цели были разработаны алгоритмы, которые находят всевозможные пути между двумя вершинами графа явных вызовов функций. Работа алгоритмов осуществляется в два этапа: 1) на подготовительном этапе строится новый граф на основе графа вызовов, из которого удаляются избыточные вершины и дуги; 2) на втором этапе происходит поиск в новом графе возможных путей от корневой вершины к заданной. Также в работе представлено экспериментальное сравнение разработанных алгоритмов с ближайшим аналогом – алгоритмом Йена.
Ключевые слова: статический анализ кода, граф вызовов, алгоритм поиска путей, алгоритм Йена, стек вызовов.

Введение

Неотъемлемой частью этапа разработки программного обеспечения является анализ кода. Данный этап необходим для раннего выявления ошибок и дефектов программ, что приводит к повышению надежности и качества кода. Одним из видов анализа является статический анализ, который позволяет найти значительное количество «простых» ошибок за приемлемое время [1]. Большинство IT компаний (в том числе и лидеры индустрии, такие как Microsoft, Google, Intel) используют инструменты статического исследования программного кода в процессе разработки. Статический анализ кода программ основан на исследовании различных представлений программ, таких как граф потока управления, граф потока данных, дерево управления и т.д. [2-4]

В данном исследовании предложены алгоритмы, основанные на анализе графа явных вызовов функций (т.к. такое представление является наиболее простым и точным). Это означает, что в анализе не рассматриваются косвенные вызовы, такие как вызов функции через

указатель или ссылку, с помощью виртуального метода и т.д. Однако алгоритм способен исследовать рекурсии (как прямые, так и косвенные).

Постановка задачи

Целью данного исследования является создание алгоритма статического анализа, для поиска всевозможных последовательностей вызовов функций, приводящих к определенной точке в программе.

Вышеописанная задача сводится к поиску в графе вызовов всех путей от корневой вершины к заданной. На подготовительном этапе строится новый граф на основе графа вызовов, из которого удаляются избыточные вершины и дуги. Под избыточными понимаются все вершины, которые не связаны с корневой или целевой вершинами. На следующем этапе происходит поиск в новом графе возможных путей от корневой вершины к заданной. Данный алгоритм похож на алгоритм поиска в глубину (deep-first search), но в отличие от последнего направлен на поиск всех, а не одного пути достижения цели [2]. Также данный алгоритм умеет обрабатывать циклические связи (что позволяет находить стек вызовов даже в программах, использующих косвенную рекурсию). При этом, циклические вызовы или рекурсии считаются одним путем с неизвестным количеством вызовов.

Разработка алгоритмов

Пусть задан граф вызовов функций G = (V,E), где V - множество вершин графа, E - множество ребер графа, a s и t - исходная и целевая вершины, между которыми мы хотим найти путь. Предположим, что в начальный момент времени все вершины графа не окрашены. Для целевой вершины *t* выполним процедуру RIN(t).

Процедура Reduce incoming nodes (параметр – вершина $u \in V$) представляет собой следующую последовательность действий:

1. Для каждой вершины w , у которой есть дуга, ведущая в вершину u выполняются следующие пункты:
   1.1. Если w уже раскрашена, то раскрашивается вершина u и осуществляется переход к п.1.
   1.2. Окрашивается вершина w . Выполняется процедура Reduce incoming nodes w.

После выполнения данной процедуры с параметром, представляющим конечную вершину t, часть вершин (которые не имеют пути в конечную вершину) отсекается. На рис. 1 представлен граф после выполнения процедуры Reduce incoming nodes (в качестве аргумента была передана вершина №11). Только вершина №8 осталась неокрашенной (т.к. не существует путей от вершины №8 к вершине №11).

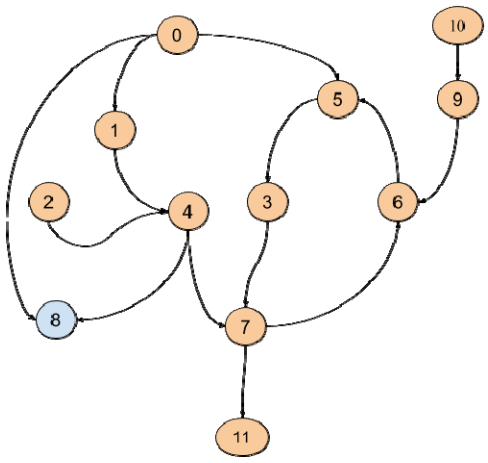

Рис. 1 - Окраска графа после выполнения процедуры RIN(11)

Далее неокрашенные вершины и дуги, инцидентные им, удаляются из графа. И выполняется процедура Reduce Outcoming Nodes с параметром s.

Reduce Outcoming Nodes (параметр - вершина $u \in V$):

1. Для каждой вершины w, в которую ведет дуга из вершины u выполняются следующие шаги:
   1.1. Если w уже раскрашена, то раскрашивается вершина u и осуществляется переход к п.1.

1.2. Окрашивается вершина *w*. Выполняется процедуру Reduce Outcoming Nodes (*w*).

После выполнения данной процедуры с параметром, представляющим начальную вершину, часть вершин (к которым не существует путей из начальной вершины) отсекается как ненужные. На рис. 2 представлен граф после выполнения процедуры Reduce outcoming nodes (в качестве аргумента была передана вершина №0). Вершины №2,9,10 остались неокрашенными (т.к. не существует путей от вершины №0 к данным вершинам).

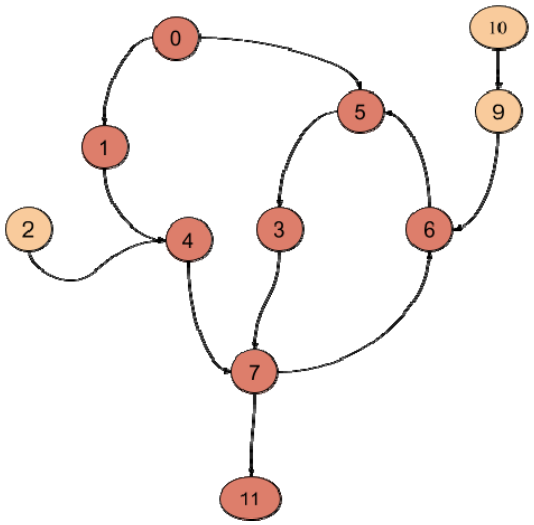

Рис. 2 - Окраска графа после выполнения процедуры RON(0)

Далее неокрашенные вершины и все дуги, ведущие к ним удаляются из графа. После выполнения этих действий получаем искомый граф.

Далее рассматривается второй этап работы алгоритма на примере. На рис. 3 представлен граф явных вызовов исследуемой программы. Допустим, существует задача найти все стеки вызовов программы до функции, отмеченной на графе вершиной номер 6.

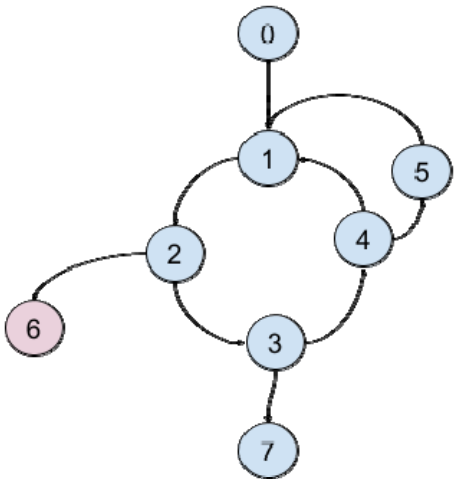

Рис. 3 - Граф явных вызовов исследуемой программы.

Возможны следующие пути от вершины 0 к вершине 6 в рассматриваемом графе:

1) Прямой путь: 0 -> 1-> 2-> 6
2) Косвенная рекурсия (первый вариант) : 0 -> (2-> 3-> 4->1)(N) -> 2->6
3) Косвенная рекурсия (второй вариант): 0 -> (2-> 3-> 4-> 5->1)(N) -> 2->6

Где N > 1 - количество проходов по заданному пути. В общем случае данное условие не может быть вычислено средствами статического анализа. Для поиска путей применяется процедура CSS (call stack searching).

В процедуру CSS (call stack searching) передаются следующие аргументы:

1) $current\_node \in V$ - текущая вершина, для которой выполняется процедура;
2) $target\_node \in V$ - целевая вершина;
3) $path = (v_1, v_2, \ldots, v_n) \in$ V x V x ... x V таких что v_i смежна с v_{i+1} для $1 \leq i < n$, и $v_1 = s$. Таким образом, *path* является путем из исходной вершины *s* до вершины, смежной с *current_node;*
4) $marked\_array \subseteq V$ - подмножество V

CSS (*current_node*, *path*, *marked_array*, *target_node*):

1. Если *current_node* = *target_node* , то *(path, current_node)* - один из искомых путей. Вернуться из процедуры.

2. Если *current_node* не имеет путей к смежным вершинам или *current_node* $\in$ *marked_array* , то происходит возврат из процедуры.

3. Создается новый массив *new_marked_array* = *marked_array*. Если *current_node* есть в *path*, то создаем новый массив $new\_marked\_array = new\_marked\_array \cup current\_node$

Для каждой смежной вершины *next_node* создается новый массив *new_path* на основе *path* с добавлением *next_node* и выполняется процедура CSS (*next_node*, *new_path*, *new_marked_array*, *target_node*).

Стоит отметить, что для получения стека вызовов функций достаточно лишь процедуры call stack searching. Алгоритмы первого этапа (reduce incoming nodes, reduce outcoming nodes) необходимы лишь для повышения эффективности поиска.

Экспериментальное исследование

С целью апробации разработанных алгоритмов было проведено экспериментальное исследование поиска путей в графе. В качестве эталонного алгоритма был взят алгоритм Йена (нахождение к-кратчайших путей) [6,7]. Алгоритм Йена позволяет находить k – кратчайших путей между заданными точками в взвешенном графе. Однако данный алгоритм не способен находить пути с циклами, поэтому в эксперименте использовались направленные взвешенные ациклические графы.

Реализация алгоритма Йена и реализация предложенного алгоритма используют одинаковую структуру хранения графа в памяти программы [8].

В результате эксперимента было обнаружено, что предложенные алгоритмы способны найти все пути, что и алгоритм Йена, но намного эффективней.

Таблица №1.

Результаты экспериментального исследования

Количество найденных путей	Время (алгоритм Йена), с	Время (предложенные алгоритмы), с	Количество вершин	Количество ребер
5832	201,8184937	0,0624004	26	68
6561	250,3192046	0,0624004	26	69
8748	503,0252245	0,1092007	29	73
10935	723,4234373	0,1092007	29	74
13122	1022,851757	0,156001	29	75
15309	1406,583017	0,1404009	29	76
17496	1842,200209	0,156001	29	77
19683	2223,341852	0,1716011	29	78
26244	4212,822605	0,312002	32	82
32805	6359,911168	0,3588023	32	83
39366	9028,573475	0,4056026	32	84
45927	13543,25802	0,4368028	32	85
52488	18353,20565	0,5148033	32	86

Результаты, представленные в табл. 1 показывают, что алгоритм Йена уступает предложенному решению. Это легко объяснить, поскольку он нацелен на поиск именно кратчайших путей (и много ресурсов расходуется на анализ наиболее выгодных путей, среди остальных). По данным таблицы 1 можно увидеть, что в некоторых случаях время, затраченное предложенными алгоритмами на поиск путей в графе более чем в 10000 раз меньше, чем время, необходимое для анализа алгоритму Йена.

Еще один недостаток алгоритма Йена более наглядно можно увидеть на представленных графиках рис. 4-5. На оси абсцисс (рис. 4 и рис. 5) показано количество путей, а ось ординат отражает величину равную количество путей деленных на количество времени (потраченного на поиск этих путей). Данную величину можно найти по формуле:

$$E = \frac{n}{\sum_i^n t_i},$$

где n – количество возможных путей, t_i - количество времени, потраченное на поиск i-го пути.

На рис. 4 видно, что производительность алгоритма Йена убывает при увеличении количества путей, в то время как предложенный алгоритм не снижает данный показатель.

Рис. 4 Результаты исследования алгоритма Йена

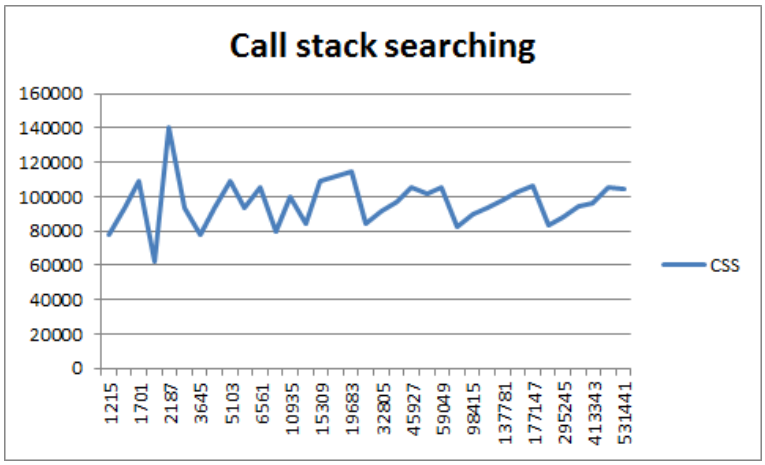

Рис. 5 Результаты исследования алгоритма CSS

Заключение

В работе были предложены алгоритмы для поиска путей между заданными точками в ориентированном графе. Целью данных алгоритмов является анализ графа явных вызовов функций и построение стека вызовов функций. Преимуществом предложенных алгоритмов является способность обрабатывать циклические связи (что позволяет находить стек вызовов даже в программах, использующих косвенную рекурсию).

Также было проведено экспериментальное исследование, показавшее значительное превосходство предложенных решений над ближайшим аналогом - алгоритмом Йена.

При одинаковом количестве найденных путей (в ациклическом направленном взвешенном графе) алгоритм Йена затратил в 10000 большее количество времени, чем предложенные алгоритмы.

Более того, результаты, полученные в ходе эксперимента, позволяют сделать вывод, что с увеличением количества путей производительность

алгоритма Йена снижается, в то время как предложенные алгоритмы показывают стабильный результат.

Разработанные алгоритмы могут быть применены в различных областях информационных технологий [9]. На их основе можно создать или дополнить инструмент для навигации по коду. Одним кликом или нажатием кнопки, программист может увидеть всевозможные стеки явных вызов функций, ведущие к заданной точке программы. Такая функциональность актуальна и представляет интерес для разработчиков программного обеспечения.

С другой стороны данный алгоритм может дополнить и повысить эффективность статического анализа. В будущих работах планируется создать методику анализа, в основе которого будут лежать данные алгоритмы.

Литература

1. Федоров А.Ю., Портнов Е.М., Кокин В.В. К вопросу использования статических анализаторов кода параллельных программ на языках C/C++ // Параллельные вычислительные технологии – XI международная конференция, ПаВТ'2017. Челябинск: ЮУрГУ, 2017. С.537.

2. Бурякова Н.А., Чернов А.В. Классификация частично формализованных и формальных моделей и методов верификации программного обеспечения // Инженерный вестник Дона, 2010. №4 URL: ivdon.ru/ru/magazine/archive/n4y2010/259/

3. Livshits B. Improving Software Security with Precise Static and Runtime Analysis // Stanford doctoral thesis, 2006. p. 229.

4. Wichmann, B. A.; Canning, A. A.; Clutterbuck, D. L.; Winsbarrow, L. A.; Ward, N. J.; Marsh, D. W. R. Industrial Perspective on Static Analysis // Software Engineering Journal. pp. 69–75.

5. Tarjan R. Depth-First Search and Linear Graph Algorithms // SIAM Journal on Computing 1. 1972. Vol. 2. pp. 146–160.

6. Yen, Jin Y. (1970). "An algorithm for finding shortest routes from all source nodes to a given destination in general networks". Quarterly of Applied Mathematics. Vol. 27, pp. 526–530

7. Yen, Jin Y. "Finding the k Shortest Loopless Paths in a Network". Management Science. 1971. Vol.17.№11. pp. 712–716

8. YenKSP // Github URL: github.com/Pent00/YenKSP

9. Берёза Н.В. Современные тенденции развития мирового и российского рынка информационных услуг // Инженерный вестник Дона, 2012. №4 URL: ivdon.ru/ru/magazine/archive/n2y2012/758

10. Ахо А.В., Лам М.С, Сети Р. Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2016. 1184 с.

References

1. Fedorov A.Yu., Portnov E.M., Kokin V.V. Parallel'nye vychislitel'nye tekhnologii XI mezhdunarodnaya konferentsiya, PaVT'2017. Chelyabinsk: YuUrGU, 2017.pp.537

2. Buryakova N.A., Chernov A.V. Inženernyj vestnik Dona (Rus), 2010. №4 URL: ivdon.ru/ru/magazine/archive/n4y2010/259/

3. Livshits B. Improving Software Security with Precise Static and Runtime Analysis. Stanford doctoral thesis, 2006. p. 229.

4. Wichmann, B. A.; Canning, A. A.; Clutterbuck, D. L.; Winsbarrow, L. A.; Ward, N. J.; Marsh, D. W. R. Software Engineering Journal. pp. 69–75.

5. Tarjan R. SIAM Journal on Computing 1. 1972. Vol. 2. pp. 146–160.

6. Yen, Jin Y. Quarterly of Applied Mathematics. Vol. 27, pp. 526–530

7. Yen, Jin Y. "Finding the k Shortest Loopless Paths in a Network". Management Science. 1971. Vol.17.№11. pp. 712–716

8. YenKSP. Github URL: github.com/Pent00/YenKSP

9. Bereza N.V. Inženernyj vestnik Dona (Rus), 2012. №4 URL: ivdon.ru/ru/magazine/archive/n2y2012/758

10. Akho A.V., Lam M.S, Seti R. Kompilyatory: printsipy, tekhnologii i instrumenty [Compilers: Principles, Techniques, and Tools]. M.: Vil'yams, 2016. 1184 p.