УДК 004.415.532.3 DOI 10.31114/2078-7707-2018-2-92-98

# Быстрый алгоритм нахождения доступных вершин графа управления при ограничениях траекторий

А.С. Щербаков

Университет Мельбурна, Австралия, andreas@softwareengineer.pro

***Аннотация* — Предлагается эвристически обоснованный алгоритм быстрого вычисления множества доступных вершин графа управления программы, допускающий задание произвольного списка “запрещенных” узлов в динамическом режиме. Алгоритм предназначен для ускорения автоматического принятия решений при моделировании и тестировании программно-аппаратных комплексов.**

***Ключевые слова* — граф управления, поиск путей в графе, тестирование программ, моделирование программ, тестирование аппаратно-программных комплексов, направленное тестирование, гамаки, кучи. Heaps.**

## I. Введение

Тестирование программного обеспечения является одной из наиболее сложных и дорогостоящих задач в процессе разработки аппаратно-программных комплексов. Несмотря на наличие технологий оптимального планирования тестов, в том числе методов и средств направленного тестирования, учитывающих граф управления программы и символические представления условий перехода в ней [1,13], ряд принципиальных недостатков ограничивают возможности достижения высоких показателей покрытия кода и поиска ошибок в ПО. В частности, разрушающие присвоения значений переменных в коде программы, наличие и последовательность которых зависит от конкретного пути выполнения [2,6,7], значительно затрудняют учет возможных ситуаций при различных входных стимулах. Такие последовательности, как правило, не учитываются явно при рассмотрении символических представлений. Доминирующий в настоящее время подход к данной проблеме основан на рассмотрении зависимости порядка выполнения операторов от присвоений значений данных как “черного” ящика, поведение которого описывается рядом сильных и слабых поведенческих гипотез, автоматически генерируемых в итеративном процессе моделирования или тестирования [3,5]. Данный способ фактически игнорирует возможность непосредственного учета участков кода, зависящих друг от друга через присвоения данных. В то же время информация о таких зависимостях может быть достаточно легко получена в процессе тестирования, например, путем одновременного динамического контроля за доступом к адресам памяти для чтения данных и исполняемого кода. Другим вариантом является инструментирование кода, которое сегодня доступно для большинства платформ. Однако использование зависимостей участков кода при планировании тестов сопряжено с существенной технической проблемой – обеспечением быстрого поиска точек ветвлений в графе управления, измерение пути выполнения в которых может приводить к существенному для тестирования изменению символической формы вычисляемых в программе выражений.

Можно сформулировать указанную проблему в более общем виде как поиск ответа на вопрос о целесообразности изменения траектории в том или ином ветвлении программы. Рассмотрение значений данных как абстрактных результатов последовательности выполнения операторов присвоения приводит к задаче о поиске в графе узлов, доступных из узла $v^+$ по траекториям, не проходящим через узлы из множества $V^-$, и, наоборот, недоступным по траекториям, проходящим из узлов $V$ не через $v^+$ [9]. При использовании алгоритмов обхода графа для принятия решения о целесообразности изменения траектории в той или иной точке ветвления необходимо затратить время, сопоставимое по порядку с временем выполнения программы, что значительно снижает применимость такого метода. В данной работе рассматривается подход к оптимизации решения данной задачи поиска, основанной на типичной структуре графа управления в виде вложенных блоков с одним входом и одним или несколькими выходами (мы будем называть их термином «***квазигамаки***»). Предлагается оптимизированный алгоритм учета, использующий там, где это возможно, типичную структуру программы в виде совокупности вложенных квазигамаков.

## II. Постановка задачи

Для простоты рассматривается функция $D(v^+,V^-)$, соответствующая одному из двух составляющих утверждений упомянутой во введении задачи поиска вершин. Для вершины $v^+$ и множества вершин $V^-$ графа управления программы $D(v^+,V^-)$ возвращает множество узлов, доступных из $v^+$ по траекториям, не проходящим ни через одну вершину, принадлежащую $V^-$. В дальнейшем изложении мы будем называть вершины, принадлежащие множеству $V^-$, вершинами-

**исключениями**. Предполагается, что число вершин-исключений сравнительно невелико (O(1)).

Сложность вычисления при каждом обращении к $D$ должна быть по возможности минимизирована в предположении случайности (и равновероятности) каждого выбора параметров. Предполагается, что вспомогательные структуры могут быть построены однократно для каждого графа управления, при этом допустимая максимальная сложность таких построений не должна превышать $O(N \log N)$, где $N$ - сумма числа ребер и графов управления. Поскольку перечисление элементов множеств само по себе может вносить сложность $O(N)$ для каждого вычисления $D$, допускается использовать сжатые представления подмножеств множества-результата (в данном случае это интервалы номеров вершин в соответствии с выбранной в рамках алгоритма нумерацией).

## III. Использование остовных деревьев для оптимизации нахождения доступных вершин

Граф управления программы, являющийся результатом компиляции исходного кода, написанного на языке программирования высокого уровня, в основном состоит из вложенных квазигамаков. Следует предположить, что, если ввести нумерацию вершин графа управления, в которой непрерывные диапазоны номеров соответствуют квазигамакам, то поиск доступных узлов в графе может быть сведен к перечислению сравнительно небольшого количества диапазонов. Однако такая задача не имеет прямого решения, так как одной вершине может соответствовать неограниченное множество квазигамаков, и нахождение оптимального разбиения которых на диапазоны представляет собой задачу оптимизации. Сложность решения такой задачи при использовании динамического программирования – кубическая по отношению к размеру графа управления[1].

Чтобы избежать неприемлемого усложнения вспомогательных построений, в настоящей работе предлагается использовать сопоставимый по эффективности вычисления $D$ жадный подход к построению иерархии квазигамаков.

Для разбиения графа управления на множество вложенных структур используется остовное дерево [8] с корнем в его входной вершине. При этом поддерево остовного дерева с корневой вершиной $v$ считается (единственным рассматриваемым) квазигамаком со входной вершиной $v$. В дальнейшем для обозначения такого поддерева используется термин ***конус*** (вершины $v$). Следует заметить, что построение остовного дерева в общем случае неоднозначно, однако, это не оказывает существенного влияния на эффективность алгоритма.

[1] Сложность однократных построений в постановке задачи не ограничена, однако уже квадратичная их сложность часто неприемлема при современных вычислительных мощностях.

Для обоснования возможности использования остовного дерева для выделения квазигамаков вначале следует показать, что квазигамак полностью покрывается конусом остовного дерева, имеющим ту же входную вершину, независимо от способа построения остовного дерева.

Теорема 1. Пусть $G$ – ориентированный граф; $r \in G$ – его вершина (вход); $T$ - остовное дерево с корнем $r$, полностью покрывающее $G$; $g$ - область графа $G$; $r \notin g$; $v \in g$ – единственная вершина, принадлежащая этой области, в которую входят ребра из (одной или более) вершин, не принадлежащих $g$. Тогда все вершины области g принадлежат поддереву дерева $T$ с корнем в вершине $v$.

***Доказательство***. Для каждой вершины $x \in g$ существует последовательность $[x_i]$ вершин, составляющих путь в дереве $T$ из $r$ в $x$ (где $i=0..k$, $x_0=r$, $x_k=x$). Так как $x$ принадлежит. $g$, а $r$ - нет, то существует вершина $x_i \in g$, в которую входит ребро из $x_{i-1}$, не принадлежащей $g$. Согласно условию теоремы, такой вершиной может являться только $v$ (так как множество ребер $T$ входит во множество ребер $G$). Таким образом, все вершины $x \in g$ являются потомками $v$ в дереве $T$, то есть принадлежат его поддереву с корнем в вершине $v$.■

Рассматривая соотношение конуса и произвольного квазигамака с той же входной вершиной, можно заметить, что конус является дополнением данного квазигамака (возможно, пустым) множеством других квазигамаков. В среднем ожидается, что (1) включаемые в конус квазигамаки подобны по статистическим свойствам; (2) для группы вложенных квазигамаков в практической программе характерно частое использования общих выходных вершин (самым распространенным примером является выход из программы или функции). Поэтому, по подобию с квазигамаком, следует ожидать малого в среднем числа внешних (не входящих в некоторый конус) вершин, соединенных ребрами графа управления с вершинами, входящими в конус. Экспериментальное подтверждение данной гипотезы являлось одной из целей настоящего исследования.

В предлагаемом алгоритме множество $R(x)$ всех доступных из вершины $x$ вершин графа управления распадается на два подмножества:

$$R(x) = I(x) \cup \xi(x), \qquad (1)$$

где $I(x)$ - множество узлов, содержащихся в конусе остовного дерева с входной вершиной $x$; $\xi(x)$ - множество узлов, не принадлежащих данному конусу, соединенных входящим рёбрами хотя бы с одной вершиной, принадлежащей данному конусу.

Если пронумеровать узлы в порядке обхода остовного дерева поиском в глубину, можно поставить в соответствие каждому конусу непрерывный целочисленный диапазон номеров. В данной работе

используется нумерация с инкрементом в фазе прямого перехода по ребру.

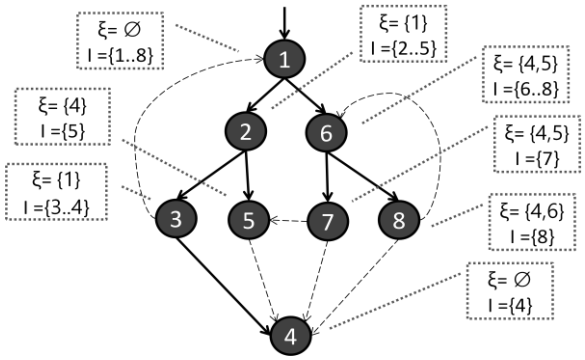


**Рис. 1. Пример построения остовного дерева на графе управления с вычисленными и для его вершин.**

Вычисление множества ξ производится на обратном проходе поиска остовного дерева в глубину путём фильтрации и объединения аналогичных множеств, вычисленных для вершин-потомков в остовном дереве. Кроме того, в ξ добавляются вершины-потомки данной вершины в графе управления, не являющиеся, однако, ее потомками в остовном дереве. При этом по мере перехода по направлению к корню остовного дерева из ξ должны быть удалены вершины, оказавшиеся внутри текущего конуса *I*. Способ эффективного построения множеств ξ описан в разделе VI. На рис. 1 представлен пример назначения вершинам графа управления значений *I* и ξ. Ребра, входящие в остовное дерево, показаны сплошными стрелками, а не вошедшие в него (и таким образом порождающие новые элементы в ξ) - штриховыми.

Вернемся к вычислению $D(x,V)$. В тех случаях, когда конус с входной вершиной $x$ не содержит узлов-исключений из $V$, достаточно использовать в его качестве $R(x)$, вычисленное в соответствии с уравнением (1). Если же конус с вершиной $x$ содержит вершину-исключение, вычисленные значения $I(x)$ и $\xi(x)$ игнорируются. Вместо этого вычисляется объединение доступных вершин по всем исходящим из $x$ ребрам, что, по сути, означает «обычный» обход графа управления без какой либо оптимизации применительно к данной вершине. В общем случае получаем

$$\mathrm{D}(x,V^-) = \begin{cases} I(x) \cup \xi(x), \text{ если } V^- \cap I(x) = \emptyset \\ \bigcup_{y \in o(x)} D(y,V^-), \text{ иначе} \end{cases} \quad (2)$$

где $o(x)$ – множество непосредственных потомков вершины $x$ в графе управления, не входящих в конус вершины $x$ в остовном дереве.

В следующем разделе рассматриваются дополнительные меры по снижению вероятности использования второй ветви уравнения (2) при практическом вычислении $D(v^+,V^-)$.

## IV. Вычисление интервалов вершин в архитектуре планирования тестов

Предполагается использование модуля, реализующего алгоритм вычисления $D(v^+,V^-)$, в комплексе с планировщиком тестов, использующим стратегию, основанную на оценке целесообразности выбора той или иной ветви потока управления в зависимости от набора входящих в $D$ вершин. Планировщик инициирует запрос на вычисление $D(v^+,V^-)$, получает результат и на его основе решает вопрос о целесообразности выбора ветви. Хотя в простейшем случае планировщику может возвращаться полностью вычисленное значение $D$, более эффективной является архитектура с приоритетной очередью интервалов номеров вершин, в которой приоритет соответствует размеру интервала, т.е. количеству вершин в нём. При этом вычисленные согласно уравнению (1) интервалы поступают в очередь без немедленной рекурсии для наследуемых вершин. По мере готовности ресурсов наибольший интервал из очереди поступает как на вход планировщика, так и на вход вычислителя $D$ (для дальнейшего перечисления составляющих $D(i)$ интервалов). При этом потребление вычислительных ресурсов значительно уменьшается благодаря первоочередной обработке наибольших интервалов и исключению покрываемых ими интервалов меньшей длины из дальнейшего рассмотрения.

## V. Использование «запасных» двоичных деревьев

В случае, если в конусе вершины $x$ оказывается вершина-исключение, как показано выше, интервал $I(x)$, вычисляемый при построении остовного дерева, неприменим, и требуется произвести итерацию вершин в глубину дерева. При этом даже одна вершина-исключение может порождать до $d$ рекурсий уравнения (1), где $d$ - глубина вершины-исключения в остовном дереве относительно $x$. Следует заметить, что остовное дерево, как правило, отнюдь не является сбалансированным, поэтому оно зачастую имеет глубину порядка общего количества узлов в нём. Применение альтернативных способов построения остовного дерева (например, с учетом весов ребер) может не приводить к положительному результату из-за наличия «узких мест» в графе управления. На рис. 2 показан пример графа, в котором из-за исключения узла №8 интервалы, ассоциированные с вершинами, становятся непригодными для вычисления $D(№j,\{№8\})$, где $j$=0...7, в результате необходимо выполнить 8 итераций и фактически произвести обход графа управления без какой-либо оптимизации. Чтобы избежать увеличения сложности предлагается использовать дополнительное двоичное дерево, построенное один раз для данного графа управления. Каждая вершина (кроме листьев) бинарного дерева соответствует диапазону номеров узлов в графе управления, а ее вершины-потомки - разбиению интервала на 2 подинтервала. Листья соответствуют узлам графа управления. Для каждой вершины

бинарного дерева вычисляются $I$ и ξ аналогично тому, как это делается для остовного дерева. Пример бинарного дерева показан на рис. 3 (соответствует примеру графа управления, показанному на рис. 2).

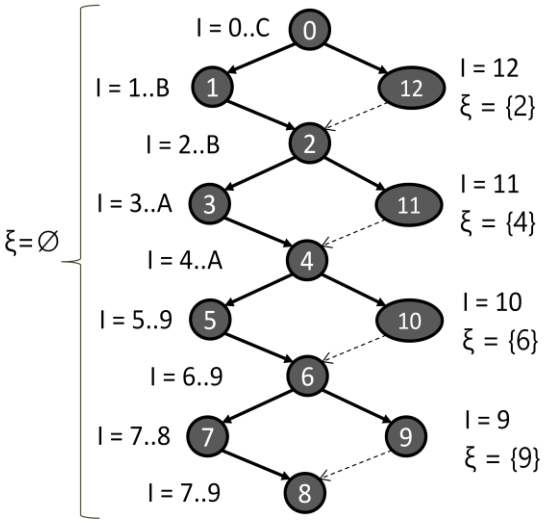


**Рис. 2. Пример построения (несбалансированного) остовного дерева в случае, когда граф управления представляет собой цепочку гамаков**

Конусу в остовном дереве соответствует в общем случае множество из не более $\log_2(K)$ непересекающихся интервалов в бинарном дереве, где $K$ - число вершин в конусе. В то же время глубина соответствующих конусу поддеревьев также ограничена логарифмом $K$. Таким образом, использование бинарного дерева для вычисления в случае присутствия вершин-исключений в рассматриваемом интервале гарантирует логарифмический верхний предел сложности.

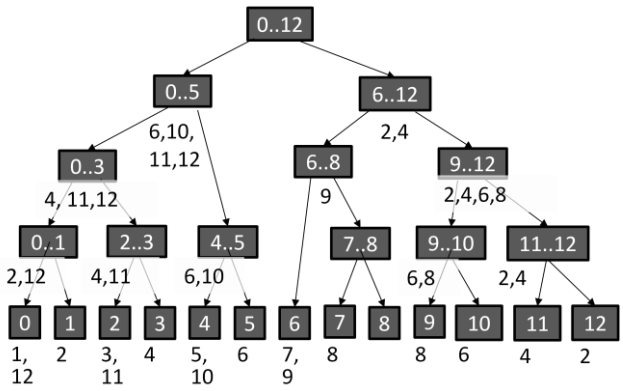


**Рис. 3. Пример построения бинарного дерева для остовного дерева графа управления (соответствует примеру, показанному на Рис. 2)**

## VI. Вычисление множеств «Внешних» узлов

В вышеприведенном алгоритме использовалось построение множеств ξ путем объединения множеств, построенных для вершин-потомков остовного дерева. Вычисление и хранение отдельных контейнеров для таких множеств внесло бы сложность порядка $O(K \cdot N)$ по времени и по размеру памяти, где $K$ - средний размер множества, $N$ - число вершин. Чтобы уменьшить сложность до ~ $O(N \cdot \log(K))$, применяется куча (*heap*) [12,13], построенная с применением совместно используемого множеством значений графа выражений [15]. Для представления экземпляра кучи, являющегося результатом операций (добавления элемента, удаления элемента или объединения куч) там, где возможно, вместо копирования структур используются ссылки на уже построенные ранее структуры. В результате сложность определяется количеством вершин графа выражений, которые следует изменить или добавить, имея в наличии граф выражений, построенный для узлов-потомков вершины графа управления. Такая сложность для сбалансированной кучи составляет $O(\log(K))$. Автором разработан код для реализации куч с графом выражений. В проведенных экспериментах сравнивается численная сложность построения множеств ξ для различных типов куч.

Укажем два важных приёма, используемых при вычислении ξ для вершин остовного дерева.

1) В используемом алгоритме мы отказываемся от исключения из кучи элементов, номера которых оказываются внутри интервала $I(x)$ при переходе от вершины-потомка к вершине-родителю в остовном дереве графа управления. Это необходимо для обеспечения пригодности ранее построенных подвыражений куч к повторному использованию. Вместо исключения мы производим последующую фильтрацию по минимальному номеру вершины в момент использования выражения для ξ, то есть используем итерацию элементов кучи с ограничением по минимальному значению элемента. Можно показать, что такая итерация с нижней границей в бинарной куче имеет сложность порядка имеющегося количества элементов, удовлетворяющих нижней границе, то есть не вносит дополнительной сложности.

2) Перед объединением ξ(a) и ξ(b), где $a$ и $b$ - потомки некоторой вершины в остовном дереве, необходимо исключить повторяющиеся элементы (чтобы элементы результата были уникальны). Это может быть сделано без вычисления пересечения куч. Достаточно, обходя остовное дерево в глубину, запоминать для каждой включаемой в $\xi(x)$ вершины $z$ соответствующую вершину $x$. Если при этом существует предыдущая $x$ (назовем ее $x'$) для той же $z$, то $z$ удаляется из аккумулированного в данный момент $\xi(h)$, где $h$ – ближайший общий предок $x$ и $x'$ в остовном дереве (поиск которого имеет сложность порядка $O(\log d)$, где $d$ - глубина остовного дерева).

## VII. Достоинства алгоритма с точки зрения интеграции

Предложенный алгоритм вычисления множеств доступных узлов обладает рядом свойств, важных для его реализации в аппаратно-программных комплексах

для тестирования систем управления или сбора данных выполненных на СБИС, в том числе в реальном времени.

1) Задействует простую и быструю независимую элементарную процедуру поиска вершин, удовлетворяющей требованиям *RESTful* интерфейсов [15], благодаря чему легко встраивается в различные параллельные и последовательные архитектурные решения, в том числе распределенные.

2) Алгоритм исключает динамическое выделение памяти переменного размера, что облегчает синхронизацию процессов и распределение ресурсов.

3) Возможна непосредственная привязка вычисляемых множеств вершин к логическим адресам памяти — отсутствие программного инструментирования обращений к памяти увеличивает производительность в 2-5 раз.

4) Возможно применение аппаратных приоритетных очередей для ускорения вычислений (при моделировании фактор возможного ускорения оценивается в ~2.5).

## VIII. Обработка вызовов

Тестируемый код обычно содержит вызовы функций, что усложняет обработку графа управления, так как вершина, в которую функция возвращает управление, зависит от вершины, в которой она вызывается. Кроме того, управление может возвращаться в различные вершины, если данная функция вырабатывает исключения или использует дальние передачи управления (в том числе прерывание или завершение работы всей программы) [16,17].

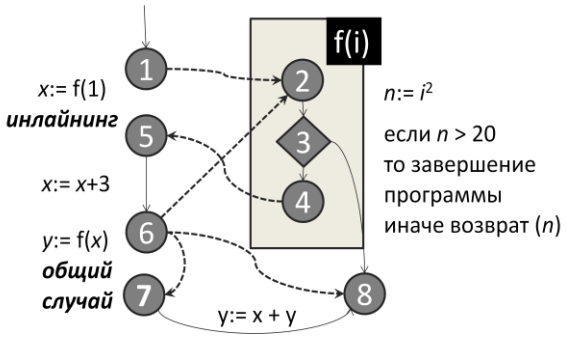


**Рис. 4. Пример обработки вызовов функции. Штриховыми линиями обозначены рёбра, относящиеся к вызовам**

**Вызов функции (инлайнинг).** Если при статическом анализе оказывается, что функция вызывается из единственной точки кода, возможно включение тела вызываемой функции в конус вершины вызова при построении остовного дерева.

**Вызов функции (общий случай).** В предлагаемом методе для каждого экземпляра вызова функции с помощью статического анализа находятся все возможные вершины-получатели управления при выходе из вызываемой функции (в том числе при возможной обработке исключений). Затем в граф управления добавляются ребра от вызывающей функцию вершины ко всем таким вершинам. Таким образом, вызов функции заменяется на условные переходы во все возможные вершины-получатели управления по выходе из нее (условия переходов считаются абстрактными). Чтобы учесть множество доступных из ее входной вершины вершин внутри ее тела (которое соответствует конусу вершины-входа функции), достаточно иметь в графе ребро, соединяющее вершину вызова с входной вершиной тела функции (рис. 4).

**Возврат из функции.** В зависимости от тактики планирования тестов при наличии в некоторой вершине оператора возврата из функции происходит вычисление D($e$,$V^-$) либо для всех возможных вершин $e$ - получателей управления (упомянутых в предыдущем пункте), либо динамический выбор одной вершины-получателя на основании имеющегося или планируемого стека вызовов. Полученные множества (рекурсивно) объединяются с $D(v^+,V^-)$.

## IX. Программная реализация

Для экспериментальной проверки гипотезы о низкой практической сложности предложенного алгоритма был создан специализированный исследовательский код [2], моделирующий решение задачи вычисления $D(v^+,V^-)$ для множества тестовых наборов $(v^+,V^-)$. Также разработан интерфейс для встраивания вычислителя $D$ в системы тестирования.

Код включает в себя сборщик и вычислитель. На входе сборщик получает промежуточные представления (LLVM IR) кода модулей тестируемой программы, генерируемые компилятором CLANG [18] из исходных кодов. После анализа шаблонов IR и сборки (linking) всех функций с известным промежуточным представлением в общий граф управления (с учетом вызовов функций) строится остовное дерево и множества внешних узлов-наследников его конусов. Вычислитель – резидентная программа (сервис), обеспечивает вычисление $D$ по запросам в параллельном режиме.

## X. Эксперименты

Для моделирования практической сложности вычисления $D(v+,V-)$ были использованы 60 приложений с открытым исходным кодом. В каждом раунде функция вычислялась для всех имеющихся в графе управления ребер $v \rightarrow u$, при этом полагалось v+, а в качестве $V^-$ использовались все прямые наследники $v$, кроме $u$. Если $V^-$ оказывалось пустым множеством, ребро исключалось из рассмотрения. Таким образом, моделировались вычисления, необходимые в случае решения задачи о целесообразности единичных альтернаций пути выполнения в каждом ветвлении тестируемого кода.

Эксперименты показывают, что среднее количество внешних по отношению к конусу остовного дерева вершин, являющихся потомками

[2] https://github.com/andreas-softwareengineer-pro/cfgsearch

вершин, покрытых данным конусом, составляет около 4.5 и не показывает существенного роста при увеличении размера конуса. В двоичном дереве такой показатель больше, около 8.5, наблюдается его примерно логарифмический рост с увеличением размера покрываемых интервалов. В среднем D вычисляется за примерно 21 итерацию при среднем размере графа управления в 1340 вершин. Таким образом, алгоритм обеспечивает достаточно низкую вычислительную сложность для использованной выборки тестируемых программ.

Также было проведено сравнение эффективности различных реализаций контейнеров (куч) для хранения множеств внешних узлов. Несмотря на отсутствие гарантированной сбалансированности и, как результат, весьма низкой ожидаемой производительности для худшего случая, куча *skew heap* [10] показывает многократно лучшее время построения и доступа, чем ближайшие по таким показателям другие исследованные типы куч [11].

## XI. Смежные подходы

Одним из известных методов организации комбинирования исполняемых участков кода с учетом зависимости условий от предшествующих присвоений переменных является партиционирование кода [4]. Основная идея метода сводится к разделению кода на участки (партиции) и присвоение номеров каждому участку так, чтобы последовательность исполнения участков с различными номерами могла быть поставлена в соответствие символическому виду выражений условий перехода. Сложность собственно партиционирования незначительна. Однако у партиционирования в таком виде есть существенный недостаток – в реальной программе партиций оказывается слишком много, и их комбинаторный перебор при тестировании не может быть осуществлен за разумное время. Такой метод пригоден лишь для разреженных связей по данным, например, если предварительно выделены “интересующие” целевые условия или применяется специально разработанная абстрактная модель потока данных. Предложенный в настоящей работе метод фактически является компромиссом между полным партиционированием кода и динамическим поиском зависимых участков по графу управления. При этом используются структурные особенности программ, написанных на языках высокого уровня.

С другой стороны, предложенный метод является компромиссом в смысле качества структур, на которые разбивается граф управления. Например, структурирование в виде гамаков [14] позволило бы уменьшить число пересечений ребрами графа управления границ конусов остовного дерева, однако в результате дублирования вершин общая ожидаемая сложность поиска оказывается худшей.

## XII. Заключение

Предложенный алгоритм поиска доступных вершин в графе управления с динамическими ограничениями служит для ускорения эффективного планирования тестов аппаратно-программных комплексов и ПО общего назначения. Малая средняя сложность алгоритма связана с типичной структурой графов управления и подтверждена экспериментально. Улучшение процедуры разбиения графа в рамках метода является предметом дальнейших исследований.

## Литература


[1] Patrice Godefroid. Compositional dynamic test generation // Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York, 2007.

[2] Uday Khedker, Amitabha Sanyal, Bageshri Sathe. Data Flow Analysis: Theory and Practice // CRC Press, 2009 .

[3] Chakrabarti A, Godefroid P. Software partitioning for effective automated unit testing // Proceedings of the 6th ACM & IEEE International conference on Embedded software, 2006. - P. 262-271.

[4] Majumdar R. and Ru-Gang Xu. Reducing test generation with information partitions // Lecture Notes in Computer Science, 2009. – V. 5643/2009. - P. 555-569.

[5] Cimatti A., Griggio A. Software model checking via IC3 // International Conference on Computer Aided Verification 2012, Berlin, Heidelberg, Springer. 2012 . P. 277-293).

[6] Allen FE, Cocke J. A program data flow analysis procedure. Communications of the ACM. 1976 Mar 1;19(3):137.

[7] Damaggio, Elio, Deutch, Alin, et Vianu, Victor. Artifact systems with data dependencies and arithmetic. ACM Transactions on Database Systems (TODS), 2012, vol. 37, no 3, p. 22.

[8] Ложкин СА. Лекции по основам кибернетики. М.: Издат. отд. Фак. ВМиК МГУ им. МВ Ломоносова; 2004.

[9] Щербаков А.С. Быстрый алгоритм учета зависимостей данных при анализе и тестировании программного обеспечения СБИС // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2016. №2. С. 76-83.

[10] Sleator DD, Tarjan RE. Self-adjusting heaps. SIAM Journal on Computing. 1986 Feb;15(1):52-69.

[11] Carlsson S, Munro JI, Poblete PV. An implicit binomial queue with constant insertion time. InScandinavian Workshop on Algorithm Theory 1988 Jul 5 (pp. 1-13). Springer, Berlin, Heidelberg.

[12] Staples J. Computation on graph-like expressions. Theoretical Computer Science. 1980 Feb 1;10(2):171-85.

[13] Patrice Godefroid. DART: Directed Automated Random Testing (joint work with Nils Klarlund and Koushik Sen) // Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation), Chicago, June 2005. - P. 213-223.

[14] Zhang, Fubo et D'Hollander, Erik H. Using hammock graphs to structure programs. // Software Engineering, IEEE Transactions on, 2004, vol. 30, no 4. - p. 231-245.

[15] Richardson L, Ruby S. RESTful web services. // " O'Reilly Media, Inc.", 2008.

[16] Grove, D., DeFouw, G., Dean, J., and Chambers, C. 1997. Call graph construction in object-oriented languages. SIGPLAN Not. 32, 10 , Oct. 1997. – P. 108-124.

[17] Callahan, D.; Carle, A.; Hall, M.W.; Kennedy, K., Constructing the procedure call multigraph. Software Engineering, IEEE Transactions on, vol.16, no.4pp.483–487, Apr 1990

[18] Lattner C. LLVM and Clang: Next generation compiler technology // The BSD Conference. 2008. P. 1-2.

# A fast Algorithm for Finding Vertices Accessible via Constrained Paths in a Control Flow Graph

A.S. Scherbakov

The University of Melbourne, Australia, andreas@softwareengineer.pro

***Abstract* — Modern techniques of directed software and firmware testing widely use symbolic exploration of conditions for program execution branch control. However, they are usually ignorant of possible influence of data value assignments to the symbolic representation of an expression. Instead, they tend to use behavioral observations for building iteratively converging sets of weak and strong hypotheses. The cause lays in the fact that although data assignment dependency information is relatively easy to extract, deriving reasonable solutions on desirable branch alternation at program branching points is a rather complicated task. One of the main challenges is high complexity of finding branch points that control interdependent assignment/usage code fragments execution patterns in a control flow graph. It requires considering graph vertex search tasks with some control vertices being restricted (in order to avoid repeating already seen paths). We propose a fast and scalable algorithm for finding sets of accessible vertices in a control flow graph being given a start vertex and a dynamically supplied list of excepted vertices. This empirically supported algorithm essentially relies on the typical structure of a program compiled from a source code. We prove experimentally that the expected complexity of each query processing is lower than O(log N), where N is total number of code fragments. The algorithm is based on substitution of a control flow graph with its spanning tree, and considering a set of vertices accessible from somewhere as a union of a vertex interval and a set of outstanding (out-of-cone) vertices. To collect the latter at a low complexity, we propose the usage of specially designed expression-graph powered version of heap containers. We also consider building (more) balanced back-off versions of spanning tree to ensure a lower complexity when walking around an excepted vertex. As well, we consider the alignment of back-off tree to its control flow graph.**

***Keywords* — control flow graph, graph traversing, software testing, software modeling, firmware testing, data dependency, variable assignments, symbolic simulation, directed testing, hammocks, heap, expression graph.**

## REFERENCES

[1] Godefroid P. Compositional dynamic test generation // Acm Sigplan Notices. 2007. V. 42. № 1. P. 47-54.

[2] Khedker U., Sanyal A., & Sathe B. Data Flow Analysis: Theory and Practice. // CRC Press. 2009. 385 p.

[3] Chakrabarti A, Godefroid P. Software partitioning for effective automated unit testing. InProceedings of the 6th ACM & IEEE International conference on Embedded software 2006 Oct 22 (pp. 262-271). ACM.

[4] Majumdar, R., Xu, R. G. Reducing test generation with information partitions // Lecture Notes in Computer Science, 2009. V. 5643/2009. P. 555-569.

[5] Cimatti A, Griggio A. Software model checking via IC3. InInternational Conference on Computer Aided Verification 2012 Jul 7 (pp. 277-293). Springer, Berlin, Heidelberg.

[6] Allen FE, Cocke J. A program data flow analysis procedure. Communications of the ACM. 1976 Mar 1;19(3):137.

[7] Damaggio, E., Deutch, A., Vianu, V. Artifact systems with data dependencies and arithmetic. // ACM Transactions on Database Systems (TODS). 2012. V. 37. № 3. p. 22.

[8] Ложкин СА. Лекции по основам кибернетики. М.: Издат. отд. Фак. ВМиК МГУ им. МВ Ломоносова; 2004.

[9] Shcherbakov A.S. A fast algorithm for data dependency tracking in Software and Firmware analysis and testing // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2016. Proceedings / edited by A. Stempkovsky, Moscow, IPPM RAS, 2016. Part2. P. 76-83.

[10] Sleator DD, Tarjan RE. Self-adjusting heaps. SIAM Journal on Computing. 1986 Feb;15(1):52-69.

[11] Carlsson S, Munro JI, Poblete PV. An implicit binomial queue with constant insertion time. InScandinavian Workshop on Algorithm Theory 1988 Jul 5 (pp. 1-13). Springer, Berlin, Heidelberg.

[12] Staples J. Computation on graph-like expressions. Theoretical Computer Science. 1980 Feb 1;10(2):171-85.

[13] Godefroid P., Klarlund N., Sen K. DART: Directed Automated Random Testing (joint work with Nils Klarlund and Koushik Sen) // Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation),. Chicago, June 2005. P. 213-223.

[14] Zhang F., D'Hollander E.H. Using hammock graphs to structure programs // IEEE Transactions on Software Engineering. 2004. V. 30. № 4. P. 231-245.

[15] Richardson L, Ruby S. RESTful web services. // " O'Reilly Media, Inc.", 2008.

[16] Grove D., DeFouw G., Dean J., Chambers C. Call graph construction in object-oriented languages // SIGPLAN Notes, V. 32. № 10. Oct. 1997. P. 108-124.

[17] Callahan D., Carle A.; Hall M.W., Kennedy K. Constructing the procedure call multigraph // IEEE Transactions on Software Engineering. Apr 1990. V. 16. №.4. P. 483–487.

[18] Lattner C. LLVM and Clang: Next generation compiler technology // The BSD Conference. 2008. P. 1-2.