УДК 51.681.3

РАЗРАБОТКА И ОБОСНОВАНИЕ АЛГОРИТМОВ НА ОСНОВЕ СЕМАНТИЧЕСКИХ СВОЙСТВ

С.Л. Крывый, А.Н. Максимец

Киевский национальный университет им. Тараса Шевченко,
03680, Киев, проспект Глушкова, 2, корпус 6,
E-mail: maksymets@gmail.com

Приводятся примеры разработки и обоснования алгоритмов на основе использования свойств предметной области. Эти свойства формулируются в виде семантических соотношений, характеризующих предметную область разрабатываемого алгоритма.

Examples of the development and validation of algorithms based on the use of the domain properties are shown. These properties are formulated in terms of semantic relationships that characterize the subject area of the developed algorithm.

Введение

Проблема разработки и обоснования алгоритмов и создания на их основе эффективных программ является одной из основных на пути построения надежного программного обеспечения. Актуальность этой проблемы особенно остро встала в последние десятилетия в связи с постоянным возрастанием сложности и объемов программного обеспечения. Такой интерес к данной проблеме связан еще и с тем, что программирование становится индустрией, в которой задействовано огромное количество людей. Одно из центральных мест среди задач решения данной проблемы занимает проблема обоснования правильности функционирования алгоритмов. Когда речь заходит о построении надежного программного обеспечения, то это значит, что оно правильно решает поставленную задачу. Однако, очень часто эффективность сильно влияет на прозрачность и понятность программного обеспечения. Это связано с тем, что эффективность, как правило, достигается за счет более глубоких и потому менее очевидных свойств предметной области. В общем случае эти проблемы решить невозможно из-за их алгоритмической неразрешимости, но ввиду их актуальности появились многочисленные направления частичного решения этой проблемы: тестирование [1], абстрактные интерпретации [2, 3], проверка на моделях [4], сетевые методы [5], логические методы (программные динамические логики и логика Хоара) [6] и т. д.

Предварительные сведения

Абстрактные типы данных. Хорошо известно, что процесс разработки программного обеспечения в упрощенном виде можно схематически представить (рисунок).

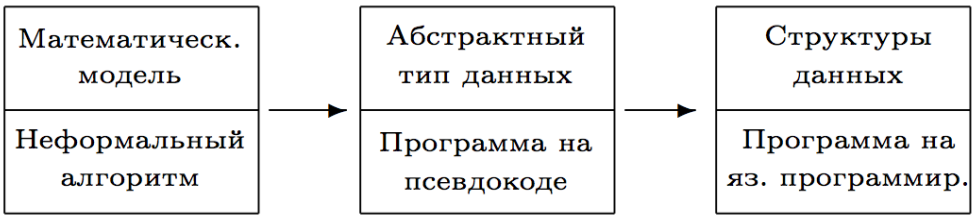

Рисунок. Схема процесса разработки

В неформальном описании алгоритма решения прикладной задачи приходится использовать типы данных присущие рассматриваемой математической модели. Данные такого типа не всегда принадлежат языку программирования и называются абстрактным типом данных **(АТД)**. В программировании различают понятия АТД, структура данных и тип данных. Под АТД понимают некоторую формальную математическую модель вместе с операциями, функциями и предикатами, определенными на этой модели. Примером такого типа данных могут служить множества, вместе с операциями объединения, пересечения и дополнения. В модели АТД операторы могут иметь операндами не только данные, которые определяются этим АТД, но и операнды языка программирования и операнды, определяемые другими АТД. Результатом выполнения оператора тоже может быть тип данных, который не определяется данным АТД. Но в рамках данного АТД предполагается, что по крайней мере один операнд или результат выполнения любого оператора имеет тип данных, определенный в данной модели АТД. Отсюда следует, что математическая модель предметной области представляет собой многоосновную алгебраическую систему.

АТД отличаются от структур данных языков программирования тем, что при использовании АТД абстрагируются от способа их реализации и учитывают только их свойства. Использование АТД позволяет разрабатывать алгоритмы и программы, базируясь на свойствах их операций и предикатов с последующим выбором наиболее эффективного способа их реализации.

Например, при построении алгоритмов манипуляции с полиномами как АТД одной из основных операций на полиномах является операция сложения полиномов и . Для реализации этой операции можно выбрать представление полиномов в памяти компьютера в виде массивов коэффициентов. Могут быть выбраны и другие интересные способы представления полиномов, но независимо от этого представления свойства операции сложения такие, как коммутативность, ассоциативность и свойства операции умножения полинома на константу, не изменяются, какой бы способ представления не был выбран. Другими известными примерами АТД являются множества, графы, матрицы, векторы, отображения и т. д.

Семантические свойства. Из приведенной на рисунке схемы следует, что процесс разработки состоит из двух этапов: на первом строится алгоритм, а на втором – программа по этому алгоритму. Учитывая это будем различать семантические свойства двух типов:

– свойства предметной области (ПО),

– свойства реализации АТД, с помощью которых построен алгоритм.

Заметим, что разделение свойств осуществляется и на этапе компиляции и отладки программ [7].

Сформулируем понятие семантического свойства, относящегося к той или иной предметной области. Семантические свойства носят первичный характер в том смысле, что они должны выполняться независимо от способов реализации АТД. Следовательно эти свойства должны согласовываться с семантикой языка программирования, в котором будет реализовываться программа. А отсюда следует, что семантические свойства ПО должны быть вычислимыми. Таким образом, приходим к такому определению.

Определение 1. Под семантическим свойством предметной области $D = D_1 \times D_2 \times \cdots \times D_n$ будем понимать соотношение $R(x, y, \ldots, z) \subseteq D$, которое выполняется (возможно при некоторых ограничениях) для любых значений переменных $x, y, \ldots, z$ из области D (удовлетворяющих этим ограничениям) и которое согласовано с семантикой реализации АТД, с помощью которых представлено это свойство.

Например, упомянутая выше операция сложения полиномов имеет семантические свойства коммутативности и ассоциативности сложения. А представление АТД "полином" в виде векторов коэффициентов со свойствами этой реализации, относятся к семантическим свойствам реализации этого АТД. Эти свойства будут другими в случае реализации полиномов значениями в заданных точках $x = c_1, y = c_2, \ldots, z = c_n$ из области D.

Учитывая, что процесс разработки алгоритмов и программ носит творческий характер, общих рекомендаций для реализации этого процесса не существует. Однако, часто из удачно сформулированного семантического свойства непосредственно вытекает алгоритм. Примерами таких свойств являются рекурсивные определения и рекуррентные соотношения. Если такие определения и соотношения найдены, то возникает вопрос об эффективности получаемого алгоритма.

Далее рассматриваются три известных алгоритма с обоснованием их правильности на основе свойств предметной области. Первый из них – это алгоритм вычисления наибольшего общего делителя двух чисел. Второй – алгоритм построения топологической сортировки, эффективность которого достигается путем перехода в другую предметную область. Третий рекурсивный алгоритм обхода неориентированного графа в глубину и его модификации на основе семантических свойств и свойств реализации АТД граф.

Алгоритм вычисления наибольшего общего делителя

Примером семантического соотношения, из которого непосредственно получается алгоритм, является семантическое свойство функции вычисления наибольшего общего делителя (НОД) двух натуральных чисел. Пусть $m, n \in N$ и существует $x \in N$ такое, что m и n кратны x. В таком случае число x называется **общим делителем** чисел m и n. Наибольший среди всех общих делителей называется **наибольшим общим делителем** чисел m и n (НОД(m, n)).

Следующие свойства отношения делимости хорошо известны:

i1) НОД(m, n) = НОД(n, m),

i2) НОД(m, n) = n, если $m = n$,

i3) НОД(m, n) = НОД($m - n, n$), если $m > n$ и

i4) НОД(m, n) = НОД($n - m, m$), если $m < n$.

Свойства **i3)** и **i4)** можно записать более компактно:

i5) НОД(m, n) = НОД($|m - n|, \min(m, n)$), если $m \neq n$.

Это соотношение и составляет одно из семантических свойств ПО N. Из него видно, что функция НОД является частично рекурсивной (она даже примитивно рекурсивная), т.е. она вычислима, и из него непосредственно следует алгоритм вычисления НОД двух натуральных чисел, написанный в псевдокоде, для этого АТД.

НОД(m, n **)**

Вход: integer m, n.

Выход: НОД (m,n).

Метод:

1. $a := m;$ $b := n;$
2. while $a \neq b$ do
   2.1. if $a > b$ then $a := a - b$ else $b := b - a$
   od
3. return (a).

Вышеприведенные свойства функции НОД гарантируют правильность этого алгоритма. Этот алгоритм эффективный в случае относительно небольших значений аргументов при использовании десятичной системы счисления. В случае больших значений аргументов более эффективным является так называемый бинарный алгоритм вычисления НОД Название этого алгоритма обосновывается тем, что как его аргументы, так и операции над ними представляются в двоичной системе счисления.

Бинарный-НОД(m,n **)**

Вход: integer m,n.

Выход: НОД (m,n).

Метод:

1. $d := 1;$
2. while m and n are even do
   2.1. ($m := m/2; n := n/2; d := 2d$);
   od
3. while $m \neq 0$ do
   3.1. while m is even do $m := m/2$; od
   3.2. while n is even do $n := n/2$; od
   3.3. $t := | m - n | /2;$
   3.4. if $m \geq n$ then $m := t$ else $n := t;$
   od
4. return ($d \cdot n$).

Этот алгоритм уже требует обоснования правильности, поскольку выполнение основных свойств НОД не очевидны. Для этого воспользуемся свойствами НОД, которые непосредственно вытекают из вышеприведенных: пусть $m = p^k m_1$, $n = p^l n_1$, где $p \neq 0$, тогда

i6) НОД(m,n) = $p^l \cdot$ НОД($p^{k-l} m_1, n_1$) = = $p^l \cdot$ НОД(m_1, n_1), если $l < k$,

i7) НОД(m,n) = $p^k \cdot$ НОД($m_1, p^{l-k} n_1$) = = $p^k \cdot$ НОД(m_1, n_1), если $k < l$.

В нашем случае $p = 2$ и после выполнения оператора 2 возникает две возможности:

а) $d = 2^l$, $m_{1'} = 2^{k-l} m_1$ и $n_{1'} = n_1$;

б) $d = 2^k$, $m_{1'} = m_1$ и $n_{1'} = 2^{l-k} n_1$.

В силу свойств **i6)** и **i7)** в обоих случаях получаем НОД($m_{1'}, n_{1'}$) = НОД(m_1, n_1).

Уничтожение множителей 2^l и 2^k выполняют операторы 3.1 и 3.2, после чего для новых значений $m_{1'}$ и $n_{1'}$ необходимо вычислить их НОД. Это вычисление выполняется в теле оператора 3.

После выполнения операторов 3.3 о 3.4 для новых значений $m_{1'}$ и $n_{1'}$ имеем:

НОД($m_{1'}, n_{1'}$)=НОД($| m_1 - n_1 |, n_1$) = НОД(m_1, n_1), если $m_1 > n_1$;

НОД($m_{1'}, n_{1'}$)=НОД($| m_1 - n_1 |, m_1$) = НОД(m_1, n_1), если $m_1 < n_1$.

Следовательно, соотношение НОД(m_1, n_1) = НОД($m_{1'}, n_{1'}$) является инвариантом цикла, который реализует оператор 3.

Остается заметить, что когда $m_{1'} = 0$, то это будет тогда и только тогда, когда $t = | m_{1'} - n_{1'} | = 0$ на предыдущем шаге, т. е. когда $m_{1'} = n_{1'}$. Но в силу **i2)** НОД($m_{1'}, n_{1'}$) = $n_{1'}$, а после выполнения оператора 4 окончательно получаем НОД(m,n) = $d \cdot$ НОД(m_1, n_1) = $d \cdot n_{1'}$, что и нужно было доказать (таблица).

Таблица

Шаг	m	n		Шаг	m	n	d
1	1424	850		1	1424	850	1
2	574	850		2	712	425	2
3	574	276		3	89	425	2
4	298	276		4	89	336	2
5	22	276		5	89	21	2
6	22	254		6	17	21	2
7	22	232		7	17	1	2
8	22	210		8	16	1	2
9	22	188		9	1	1	2
10	22	166		10			НОД (m,n) = 2
11	22	144					
12	22	122					
13	22	100					
14	22	78					
15	22	56					
16	22	34					
17	22	12					
18	10	12					
19	10	2					
20	8	2					
21	6	2					
22	4	2					
23	2	2	НОД(m,n)=2				

Пример 1. Приведем результаты вычисления НОД первым и вторым алгоритмами в предположении представления чисел в двоичной системе исчисления.

Первый алгоритм требует выполнения 46 сравнений и вычитаний, а второй – 18 сравнений и вычитаний, поскольку деление на 2 в двоичной системе представления чисел выполняется за константу, которая не зависит от величин чисел-аргументов.

Алгоритм топологической сортировки

Рассмотрим второй пример алгоритма, который использует другие АТД для своей реализации. Это алгоритм топологической сортировки, который вкладывает частичный порядок в линейный. В отличие от предыдущего алгоритма для этого алгоритма нельзя сформулировать семантическое свойство, из которого бы он вытекал. Для того чтобы сформулировать семантические свойства ПО, необходимо выходить в область теории графов. Приведем необходимые пояснения.

Определение 2. Отношение линейного порядка $\le$ на конечном множестве A называется топологической сортировкой отношения частичного порядка $\le_1$ на этом множестве, если для любых элементов $a,b \in A$ из $a \le_1 b$ следует $a \le b$.

Из определения вытекает такое семантическое соотношение $a \le_1 b \to a \le b$, используя которое можно построить такой алгоритм топологической сортировки: найти минимальный элемент в множестве A и занести его в список линейно упорядоченных элементов; удалить минимальный элемент из множества A и повторить предыдущий шаг, если оставшееся множество A не пусто. Однако, анализ сложности такого алгоритма показывает его низкую эффективность с точки зрения времени выполнения. Поиск более эффективной реализации приводит к необходимости перехода в другую предметную область – область ациклических графов. Известно, что отношение частичного порядка на конечном множестве A представляется в виде диаграммы Гассе, которая

является ациклическим орграфом $G = (V, E)$ (DAG – directed acyclic graph). Тогда топологическая сортировка сводится к линейному упорядочению вершин ациклического орграфа таким образом: если существует дуга из вершины v в вершину u в графе G, то в искомом линейно упорядоченном списке вершин орграфа вершина v предшествует вершине u. Это и есть трансформация семантического отношения $a \le_1 b \to a \le b$ в терминах нового АТД, которым является ациклический граф.

В терминах этого АТД алгоритм топологической сортировки принимает вид:

```
TOPSORT( G = (V, E) )
begin
 for each v ∈ V do
   build In(v) = {u ∈ V | (u, v) ∈ E};
 od
 V' := V;   V'' := ∅;
 while V' ≠ ∅ do
    take u from V';
    if In(u) = ∅ then
       V' := V' \ {u};   V'' := V'' ∪ {u};
       for each v ∈ V do
           In(v) := In(v) \ {u};
       od
    else (STOP: G is not DAG) (* добавленная часть *)
    fi
 od
end
```

Независимо от того, как будут реализованы множества, фигурирующие в этом алгоритме, сам алгоритм остается неизменным. Но реализация АТД "множество" сильно влияет на эффективность данного алгоритма. Простой анализ показывает, что множества V' и V'' следует реализовывать в виде очередей. Причем, вершины с пустыми множествами $In(v)$ лучше размещать на одном из концов очереди, а множества $In(v)$ реализовывать с помощью хеш-таблиц.

Покажем правильность алгоритма и приведем оценку его временной сложности.

Пусть множества V' и V'' реализованы в виде очередей, причем вершина $v \in V'$ переносится на левый конец очереди V', как только $In(v)$ становится пустым множеством. Выбор элементов из очереди V' выполняется слева, а добавление элементов в очередь V'' выполняется с правого конца.

Для доказательства правильности алгоритма необходимо показать:

а) что когда $v \le_1 u$ (т. е. из вершины v к вершине u существует путь в графе G), то в множестве V'' вершина v предшествует вершине u;

б) что все вершины графа G присутствуют в списке V''.

Доказательство пункта а) достаточно очевидно, поскольку если $v \le_1 v$, то это значит, что $In(u)$ включает вершину v. Но тогда вершина u не попадет в список V'' до тех пор, пока из множества $In(u)$ не будут удалены все вершины (а, значит, и вершина v). Следовательно, вершина v появится в списке V'' раньше вершины u, а это значит, что $v \le u$.

Доказательство пункта б) связано с анализом ситуации, когда некоторая вершина u не попадает в список V''. Это возможно только тогда, когда $In(u) \ne \varnothing$, что значит наличие цикла в исходном графе G. Действительно, если $In(u) \ne \varnothing$, то существует вершина v в графе G принадлежащая к $In(u)$ и не удаляемая из этого множества в процессе выполнения алгоритма. Но это означает, что и $In(v) \ne \varnothing$ и тогда вершины u и v связаны между собой маршрутом, который образует цикл.

Таким образом, для правильности работы алгоритма *TOPSORT* необходимо внести в алгоритм проверку ацикличности исходного графа. При заданной организации выбора элементов из очереди V' из этого множества всегда выбирается вершина u с пустым множеством $In(u)$. Если такого элемента не окажется, то это и будет означать наличие цикла в исходном графе. С этой целью в алгоритм добавлена часть *else* в условный оператор, входящий в тело цикла *while*.

Пример 2. Применим этот алгоритм к решению такой задачи. Пусть a, b, c, d, e, f, g, h означают курсы, которые должны быть прочитаны лекторами кафедры.

Порядок зависимости курсов определяется с помощью ациклического орграфа $G = (V, E)$, который показан на рис. 1. Необходимо найти последовательность чтения курсов с учетом их зависимости.

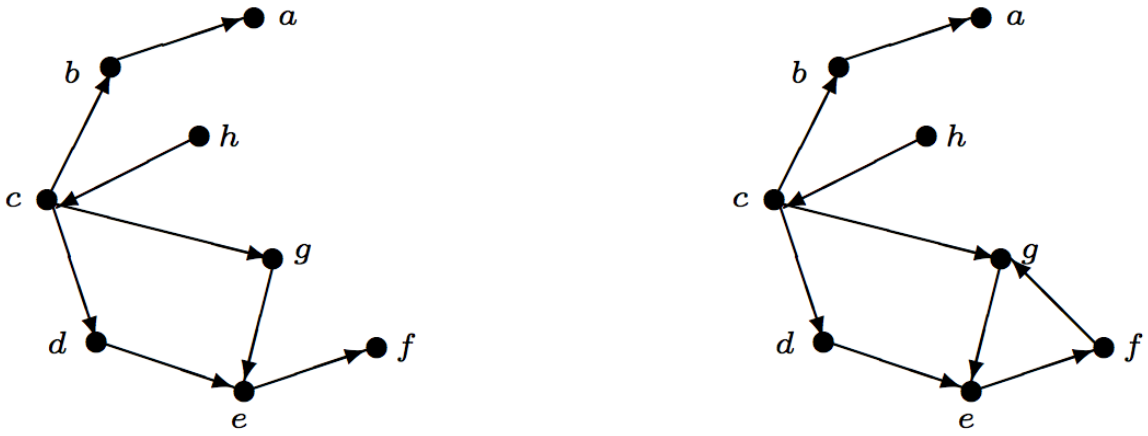

Рис. 1. Порядок зависимости курсов

После выполнения оператора *for* получаем такую последовательность множеств $In(v)$:

$$In(h) = \varnothing,\ In(c) = \{h\},\ In(b) = \{c\},\ In(a) = \{b\},$$

$$In(d) = \{c\},\ In(g) = \{c\},\ In(f) = \{e\},\ In(e) = \{d, g\}.$$

Согласно расположению элементов в очереди V' первым элементом в ней всегда должна быть вершина v, у которой $In(v) = \varnothing$. Дальнейшее выполнение алгоритма TOPSORT дает такую последовательность чтения курсов: h, c, b, d, g, a, e, f.

Это одна из возможных последовательностей, поскольку на некоторой итерации цикла *while* возможен выбор. Например, после обработки вершин h и c, кандидатами на выбор стают вершины b, d, g (алгоритм выбрал вершину b).

Если применить модифицированный алгоритм *TOPSORT* к орграфу G_1, то он обнаруживает что орграф G_1 не ациклический.

Алгоритм обхода графа в глубину

Рассмотрим третий способ построения алгоритмов с помощью рекурсии, а именно – алгоритм обхода вершин графа в глубину. Этот алгоритм хорошо известен и называется DFS-алгоритмом (DFS – depth first search).

Перед представлением этого алгоритма, напомним некоторые определения из теории графов. Неориентированный граф называется связным, если между произвольными его вершинами существует маршрут. Очевидно, что в случае неориентированных графов отношение "между вершинами существует маршрут" является отношением эквивалентности. Тогда за этим отношением множества вершин V и ребер E разбиваются на классы эквивалентности $V_1, V_2, \ldots, V_k$ и $E_1, E_2, \ldots, E_k$ такие, что пары $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2), \ldots, G_k = (V_k, E_k)$ образуют связные подграфы графа $G = (V, E)$. Эти подграфы называются **компонентами связности** графа G и задача проверки связности графа состоит в том, чтобы найти количество этих компонент.

Работа алгоритма сводится к тому, что сначала все вершины графа $G = (V, E)$ размечаются как не пройденные (для этого служит массив *visited* – массив логических размерности $|V|$), строятся списки смежных вершин (список $Adj(v)$ вершины v). Дальше обработка вершин графа алгоритмом начинается с произвольной вершины, которая считается начальной. При первом попадании в вершину она размечается как пройденная (*visited*) и к ней рекурсивно применяется *DFS*-алгоритм.

В псевдокоде *DFS*-алгоритм имеет вид:

$DFS(G)$

Вход: граф $G = (V, E)$, где $|V| = n$ и $Adj(v)$ – множество вершин, смежных с вершиной v.

Выход: граф G со всеми помеченными вершинами.

Метод:

```
for v := 1 to n do
  visited(v) := 0
od
for v := 1 to n do
  if visited(v) = 0 then
    Dfs(v)
  fi
od
 Procedure Dfs(v)
  visited(v) := 1;
  for each w ∈ Adj(v) do
    if visited(w) = 0 then
      Dfs(w)
    fi
  od
 End Dfs
```

Для доказательства правильности работы данного алгоритма необходимо знать представление графа в памяти компьютера. Наиболее часто используется представление графа в виде матрицы смежности или списками смежности. Более экономным с точки зрения используемой памяти является представление графа списками смежности.

Например, граф G из рис. 2 имеет представление в виде мультисписка смежных вершин приведено на рис. 3.

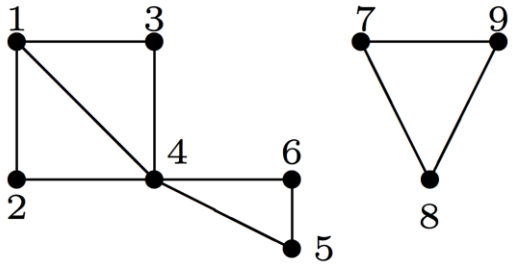

Рис. 2. Несвязный граф

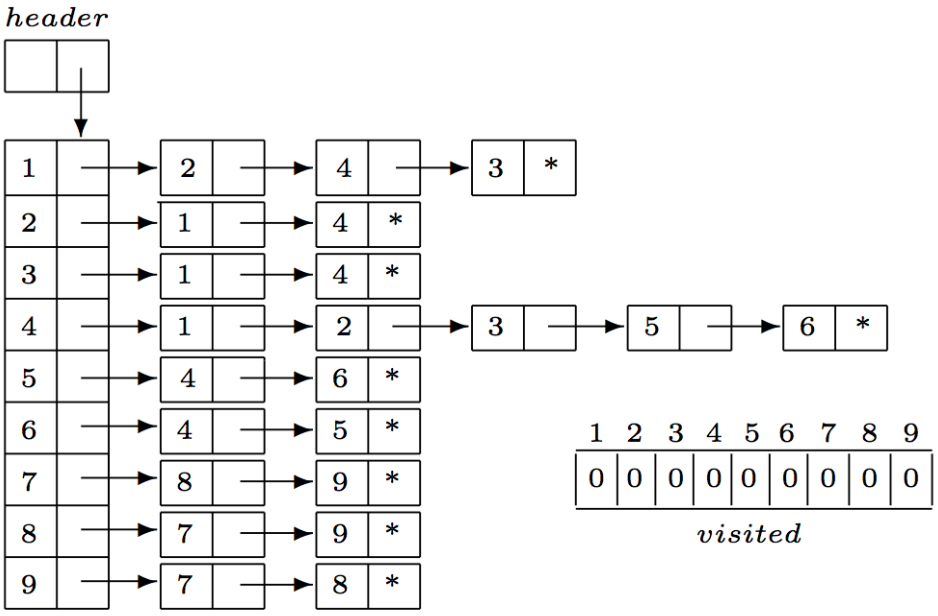

Рис. 3. Представление графа мультисписками смежности

Дальше предполагается, что граф представляется в виде мультисписков смежностей. Для удобства первый список, включающий все вершины орграфа, будем называть вертикальным списком, а списки смежных вершин – горизонтальными списками. Заметим, что операции удаления ребра и введение ребра сводятся к удалению и добавлению соответствующего элемента в горизонтальном списке смежности соответственно. Операции введения вершины и удаления вершины в этом случае сводятся к добавлению в конец вертикального списка нового элемента и удаления всего горизонтального списка смежности данной вершины.

Доказательство правильности работы *DFS* -алгоритма сводится к доказательству наличия пометок у всех вершин графа. В силу рекурсивности данного алгоритма достаточно доказать правильность одного рекурсивного вызова и корректность завершения этого вызова.

Пусть граф G связный, тогда, начиная с начальной вершины G, *DFS* -алгоритм начинает с разметки всех его вершин как не помеченные путем присваивания значения 0 всем вершинам (для этого служит первый цикл for и массив *visited*).

Второй цикл, начиная с начальной вершины вертикального списка рекурсивно обрабатывает эту вершину при условии её непомеченности. Эта обработка начинается с пометки вершины в массиве *visited* и рекурсивной обработки первой непомеченной вершины w, смежной с вершиной v. Здесь возможны два случаи: а) вершина w не имеет смежных вершин, кроме вершины v и б) вершина w смежна не только с v, но и с некоторой вершиной $u \neq v$.

В случае а) *Dfs* -процедура заканчивает обработку вершины w, поскольку $Adj(w) = \{v\}$ и v уже была помечена. Рекурсия возвращается к следующей непомеченной вершине w', смежной с вершиной v. А это означает, что вершина w была помечена и к ней *Dfs* -процедура уже не будет применяться. Что говорит о том, что обработка была выполнена правильно.

В случае б) *Dfs* -процедура начинает рекурсивную обработку вершины u, присваивая ей пометку 1 в массиве *visited* . Теперь ситуация с обработкой u повторяется в точно так, как и с вершиной w. В результате вершина u получает пометку, что свидетельствует о правильности работы *DFS* -алгоритма.

Если *DFS* -алгоритм применить к графу из рис. 2, то *Dfs* -процедура порождает путь 1,2,4,3,5,6 и заканчивает свою работу. Однако, цикл for *DFS* -алгоритма выполнит очередной вызов *Dfs* -процедуры, которая начнет обработку вершины 7. Эта ситуация означает существование новой компоненты связности. Отсюда следует, что можно модифицировать *DFS* -алгоритм так, чтобы он находил количество компонент связности и указатели на начальные вершины этих компонент.

В результате выполнения *DFS* -алгоритма генерируются *DFS* -деревья, которые соответствуют компонентам связности (для сохранения *DFS* -дерева служат списки $parent(v)$), а значение переменной c указывает на количество этих компонент.

Модифицированный *DFS* -алгоритм. Таким образом, модифицированный алгоритм принимает вид:

$DFS\text{-}CONNECTED\text{-}COMPONENT(G)$

begin

1. $c := 0$;
2. *for all* $v \in V$ *do*;
3. $visited(v) := 0$;
4. $parent(v) := NIL$; /* NIL - пустой список */
5. *od*;
6. *for all* $v \in V$ *do*;
7. *if not* $visited(v)$ *then*
8. $c := c + 1$;
9. $DFS(v, c)$;
10. *fi*;
11. *od*;

end

$DFS(v, c)$

begin

1. $visited(v) := 1$;
2. $component(v) := c$;
3. *for all* $w \in Adj(v)$ *do*;

4. *if not visited*(*w*) *then*
5. *parent*(*w*) := *v*;
6. *DFS*(*w*,*c*);
7. *fi*;
8. *od*;

end

Если в результате выполнения алгоритма *DFS - CONNECTED - COMPONENT* получаем $c = 1$, то исходный граф $G = (V, E)$ связный, иначе – несвязен.

Пример 3. Пусть $G = (V, E)$ – граф, приведен на рис. 2. Если начальной вершиной есть 1, то значение $c = 2$ (граф G несвязен) и деревья, порождаемые *DFS* -алгоритмом, имеют вид (рис. 4).

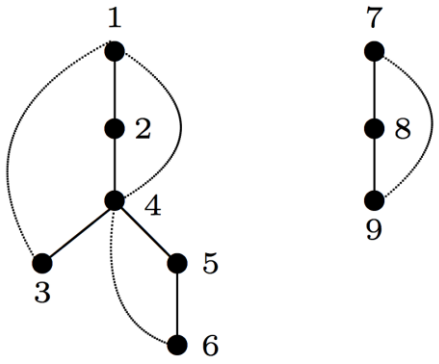

Рис. 4. Деревья, порождаемые *DFS* -алгоритмом ($c = 2$)

Известно, что временная сложность *DFS* -алгоритма выражается величиной $O(|V|+|E|)$.

Заметим, что *DFS* -деревья (указатели на начальные вершины этих деревьев находятся в списке *component*(*v*)), которые строит *DFS* -алгоритм, зависят от порядка обхода вершин, смежных с текущей вершиной. Отсюда следует, что не всегда некоторое ребро будет ребром такого дерева. Ребра, которые принадлежат *DFS* -дереву, называются прямыми ребрами, а остальные рёбра графа называются **обратными**. Очевидно, что при помощи *DFS* -алгоритма можно находить не только прямые, но и обратные рёбра в процессе обхода его вершин. Выше на рисунке выпуклыми кривыми показаны обратные рёбра $(3,1),(4,1),(5,4)$ и $(9,7)$, найденные *DFS* -алгоритмом.

Вывод

Отметим, что необходимость поиска семантических свойств предметной области возникает в связи с тем, что информации, извлеченной из текста алгоритма или программы (например, в виде инвариантных соотношений), как правило, недостаточно для обоснования их правильности. Многие свойства, особенно семантического характера, явно не фигурируют в алгоритме и тем более в программе его реализации. Поиск семантических свойств сопряжен с трудностями как теоретического, так и практического характера и его успех зависит прежде всего от индивидуальных качеств разработчика программного обеспечения. В этом, наверное, и состоит главная сложность проблемы построения качественного и надежного програмного обеспечения. Такой подход к построению и обоснованию эффективных алгоритмов описывался в работах [9, 10].

1. *Калниньш А.А., Борзов Ю.В.* Инвентаризация идей тестирования программ. Рига: ЛГУ им. Стучки. – 1981. – 54 с.
2. *Cousot P.* Abstract Interpretation Based Formal Methods and Future Challenges. http://www.di.ens.fr/ cousot/. – 2003. – P. 131–151. Chap. 10. – P. 303–342.
3. *Cousot P.* Verification by abstract Interpretation. – Lecture Notes in Comp. Science. – 2003. – N 2772. – P. 243–268.
4. *Clarke E.M., Grumberg Jr. O., Peled D.* Model Checking. The MIT Press: Cambridge, Massachusetts, London, England. – 2001. – 356 p.
5. *Penczek W., Pólrola A.* Advanced in Verification on Time Petri Nets and Timed Automata. Springer-Verlag Berlin Heidelberg. – 2006. – 258 p.
6. *Ben-Ari M.* Mathematical Logic for Computer Science. Springer-Verlag London Limited. – 2001. – 345 p.
7. *Nielson F.* Two-level semantics and abstract interpretation // Theor. Comp Scien. (Fundamental Studies). – 1989. – N 69. – P. 117–242.
8. *Кривий С.Л.* Вступ до методів створення програмних продуктів. Чернівці: "Букрек". – 2012. – 424 с.
9. *Годлевский А.Б., Кривой С.Л.* Трансформационный синтез эффективных алгоритмов с учетом дополнительных спецификаций // Кибернетика. – 1986. – № 6. – С. 34–43.
10. *Годлевский А.Б., Кривой С.Л.* О проектировании эффективных алгоритмов приведения автоматов для некоторых отношений эквивалентности // Кибернетика и системный анализ. – 1989. – № 6. – С. 54–61.