И.Б.Бурдонов.
НЕИЗБЫТОЧНЫЕ АЛГОРИТМЫ ОБХОДА ОРИЕНТИРОВАННЫХ ГРАФОВ. НЕДЕТЕРМИНИРОВАННННЫЙ СЛУЧАЙ.
"Программирование". 2004. No. 4.

Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированнный случай.

И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин

1. Введение

Данная статья является продолжением статьи [18], которая была посвящена обходу детерминированных графов как основе тестирования конечных автоматов (точнее, объектов, рассматриваемых как конечные автоматы) по спецификациям.

Автомат определяется множеством своих состояний и переходов. Переход автомата – это четверка **(v,x,y,v`)**, где **v** – пресостояние, **x** – стимул, **y** – реакция, **v`** – постсостояние. Обычно автомат задается ориентированным графом состояний, вершины которого – состояния, а дуги – переходы. Автомат (граф состояний) всюду определен, если в каждом состоянии **v** допустим каждый стимул **x**, то есть, существует хотя бы один переход вида **(v,x,y,v`)**. В противном случае, автомат частично определен. Автомат (граф состояний) детерминирован, если пресостояние и стимул однозначно определяют реакцию и постсостояние. В предыдущей статье мы рассматривали детерминированные частично определенные автоматы, а здесь исследуем недетерминированный случай.

Если граф состояний автомата известен, то обычно интересуются вопросами, удовлетворяет ли он тем или иным требованиям. Эти задачи решаются аналитическими методами и о тестировании обычно речь не идет. Тестирование нужно тогда, когда граф состояний автомата неизвестен. Автомат рассматривается как «черный ящик»: мы можем подавать на автомат стимулы и получать ответную информацию о выполненном переходе, то есть, в общем случае, о реакции и постсостоянии. Задачей тестирования является проверка того, что тестируемый автомат удовлетворяет заранее заданным спецификационным требованиям. Это тестирование соответствия (*conformance testing*) в широком смысле. В общем случае спецификация не подразумевает проверки каждого перехода автомата. Если, например, мы хотим проверить, что число состояний автомата не меньше заданного, то тестирование прекращается, как только мы в этом убедимся, и оставшиеся непроверенными переходы нас не интересуют. Однако, такие требования являются скорее исключением, чем правилом. Обычно нас интересует полная функциональность автомата, и нам требуется проверка каждого его перехода, если это возможно. Такое тестирование опирается на следующие предположения.

Изменение состояния. Состояние меняется только в результате тестового воздействия, то есть, подачи стимула на автомат. С одной стороны, это означает, что автомат находится под исключительным управлением теста и никто «не мешает» тестированию, точнее, такое постороннее воздействие не изменяет наблюдаемую функциональность автомата. С

другой стороны, это означает, что у теста нет других средств изменить состояние автомата.

Допустимость стимулов. В каждый момент времени мы можем тем или иным способом узнать, какие стимулы можно подавать на автомат. При тестировании могут использоваться не все стимулы, допускаемые в реализации. Фактически, это означает, что для реализации **R** и модели **M** тестируется подавтомат **R(M)⊆R**, определяемый переходами по модельным стимулам и состояниями, достижимыми из начального по таким переходам.

Наблюдаемость реакций. После подачи стимула на автомат мы может наблюдать выдаваемую им реакцию. Собственно говоря, при тестировании мы проверяем как раз эту реакцию. В противном случае, автомат совершал бы какие-то переходы, но мы не смогли бы узнать, правильные они или нет. С другой стороны, о правильности переходов можно судить также по постсостояниям при условии, что они наблюдаемы.

Отдельно стоит вопрос о наблюдаемости состояний автомата. Если в любой момент времени мы можем узнать состояние автомата, прочитав его или получив в ответ на специальную операцию *status message* (предполагается, что операция не изменяет состояние) [9], то такое тестирование будем называть тестированием с открытым состоянием. В противном случае, будем говорить о тестировании со скрытым состоянием.

Специальный случай тестирования соответствия (в узком смысле), когда спецификация – это модельный автомат, эксплицитно заданный своим графом состояний, и проверяется эквивалентность реализационного (тестируемого) автомата модельному [9]. Два состояния (одного или разных автоматов) эквивалентны, если любая последовательность стимулов, допустимая, начиная с одного состояния, допустима, начиная с другого состояния, и вызывает одну и ту же последовательность реакций. Автоматы эквивалентны, если каждому состоянию одного автомата соответствует эквивалентное ему состояние другого автомата. Модельный автомат, тем самым, описывает класс эквивалентных ему реализационных автоматов.

Здесь можно выделить три основных проблемы:

1) Проблема недетерминизма. В общем случае мы никогда не можем быть уверены, что проверили все переходы реализационного автомата. Для решения этой проблемы иногда вводят специальные допущения [9]. Например, предполагают, что при достаточно большом числе проходов из данной вершины **v** по данному стимулу **x** будут пройдены все переходы вида **(v,x,y,v`)**. Или вводят вероятности переходов и проводят достаточное количество испытаний для обеспечения достаточной вероятности полноты теста. Другой подход основан на введении *тестовой эквивалентности* переходов, когда считается достаточным проверить хотя бы один переход из класса эквивалентности. Если таким классом считать все переходы из данного состояния по данному стимулу, то при тестировании нам достаточно проверить каждый стимул в каждом состоянии. Это является аналогом обхода графа, который мы будем называть *обходом по стимулам*.

2) Проблемы полноты тестирования. Как с помощью тестирования проверить эквивалентность модельного и реализационного автоматов? Для детерминированного случая имеются хорошо разработанные методы решения этой проблемы. При тестировании с открытым состоянием задача сводится к обходу модельного графа [9], то есть, построении маршрута, проходящего по всем модельным переходам. Двигаясь по маршруту, мы для каждого модельного перехода **(v,x,y,v`)** подаем на реализационный

автомат тот же стимул **x** и проверяем, что получаемые реализационные реакция и постсостояние совпадают с модельными (**y,v`**). Если реализационное состояние нам недоступно (тестирование со скрытым состоянием), приходится вводить специальные ограничения на реализацию и модель и прибегать к гораздо более сложным методам проверяющих последовательностей (*checking sequence*) [9]. В недетерминированном случае задача осложняется проблемой недетерминизма. Однако, по крайней мере, обход по стимулам должен выполняться всегда, хотя он недостаточен даже при тестировании с открытым состоянием (если только мы не ограничиваемся проверкой одного перехода в каждом состоянии по каждому стимулу).

3) Проблема имплицитности спецификаций. К сожалению, на практике спецификации, во-первых, не описывают явно модельный автомат и существует проблема его эксплицирования, а, во-вторых, реализация должна быть эквивалентна не полному модельному автомату, а некоторому его подавтомату, заранее неизвестному. Наиболее широко распространенный случай – это имплицитные спецификации в виде пред- и постусловий. Предусловие – предикат над пресостоянием и стимулом – определяет допустимость стимулов в состояниях, а постусловие – предикат над пресостоянием, стимулом, реакцией и постсостоянием – определяет возможные переходы. Эксплицирование автомата из таких спецификаций сводится к решению системы уравнений общего вида и, в общем случае, не имеет удовлетворительного решения. Но дело не только в этом.

Обычно считается, что спецификация описывает *возможные*, но *не обязательные*, переходы автомата. Точнее, если спецификация допускает несколько переходов по данному стимулу из данного пресостояния, то требуется, чтобы реализация имела хотя бы один из таких переходов, но не обязательно все. Фактически, это означает, что модельному автомату соответствует не один класс эквивалентных реализационных автоматов, а семейство таких классов. Реализационный автомат, точнее, как сказано выше, его подавтомат $\mathbf{R(M)}\subseteq\mathbf{R}$, определяемый модельными стимулами, эквивалентен некоторому подавтомату **M(R)** спецификационного автомата **M**, причем в каждом состоянии **M(R)** допустимы все стимулы, которые в этом состоянии допустимы в **M**, но среди всех переходов в **M** из данного состояния по данному стимулу не все должны присутствовать в **M(R)**. (Иногда в этом случае говорят о *квази-эквивалентности* **R** и **M(R)** и *редукции* **R** к **M** [6].) Понятно, что в этом случае, во-первых, работа по эксплицированию спецификационного автомата **M** может оказаться чрезмерной, так как нам требуется только его подавтомат **M(R)**, число состояний и переходов которого может быть во много раз меньшим, чем в **M**. Во-вторых, сам **M(R)** заранее неизвестен, поскольку определяется не только **M**, но и **R**.

Проблемы построения маршрутов в графе состояний конечного автомата известны уже давно. Они были сформулированы для маршрутов некоторых специфических классов еще на самой заре теории конечных автоматов. Время от времени проблемы такого рода привлекали интерес исследователей в связи с проблемами тестирования, основанного на модели конечного автомата [3,8,7]. Различные методы тестирования на основе недетерминированных моделей активно изучаются, см. работы Петренко и др. [5] или ASM группы компании Microsoft Researh [16]. Обычно предполагаются существенные ограничения на поведения реализации. Например, требуется, чтобы реализация была детерминирована, как в работе Петренко, Евтушенко и Bochmann [10], или чтобы выполнялись сложные тестовые допущения, которые предполагают, что все возможные переходы могут быть получены после некоторого числа попыток, см. [4, 2].

В части 2 настоящей статьи предлагается «рамочный» подход к решению этих проблем. Выделяется первая фаза тестирования по имплицитным спецификациям, задача которой – эксплицирование модельного подавтомата, пригодного для использования на следующих фазах (точнее, такого подграфа его графа состояний, который содержит все достижимые состояния и хотя бы один переход для каждого стимула в каждом состоянии, но, быть может, не все переходы). Эта задача сводится к обходу по стимулам неизвестного недетерминированного графа. Соответствующие алгоритмы, получающие информацию о графе в процессе его обхода по стимулам, мы называем *неизбыточными*.

В части 3 формулируются основные понятия графа, обхода по стимулам и алгоритма, в части 4 изучается проблема существования и длины такого обхода, а в части 5 предлагаются конкретные неизбыточные алгоритмы.

2. Тестирование недетерминированного автомата

Будем считать, что модель **M** задана имплицитной спецификацией с помощью пред- и постусловий. Предусловие задает допустимость каждого стимула **x** в каждом состоянии **v** модели: **PRE(v,x)=true**. Постусловие задает допустимость полученных реакции **y** и постсостояния **v`** при переходе из пресостояния **v** по стимулу **x**: **POST(v,x,y,v`)=true**.

Мы предлагаем рассматривать тестирование на основе имплицитных спецификаций как двухфазное. Задача первой фазы – эксплицирование подавтомата $\mathbf{M(R)}\subseteq\mathbf{M}$ для реализации **R**, задача второй фазы – тестирование реализационного подавтомата $\mathbf{R(M)}\subseteq\mathbf{R}$ по модельному автомату **M(R)**.

Сначала рассмотрим, как определяются автоматы **R(M)** и **M(R)** в процессе тестирования с *открытым* состоянием. Предполагается, что начальное состояние $\mathbf{v_0}$ модели **M** такое же как начальное состояние реализации **R** (что можно проверить с помощью *status message*) и мы помещаем его в **R(M)** и **M(R)**. Подаем на реализацию стимул $\mathbf{x_0}$, допустимый в **M** в состоянии $\mathbf{v_0}$. Таким стимулом является любое решение уравнения предусловия спецификации $\mathbf{PRE(v_0, x_0)=true}$. Реализация выполняет некоторый переход $\mathbf{(v_0,x_0,y_0,v_1)}$, который добавляется в **R(M)**. Получая реакцию $\mathbf{y_0}$ и постсостояние $\mathbf{v_1}$, проверяем, что в **M** существует переход $\mathbf{(v_0,x_0,y_0,v_1)}$, то есть, проверяем постусловие спецификации $\mathbf{POST(v_0,x_0,y_0,v_1)=true}$. Если постусловие не выполнено, фиксируем ошибку в реализации. В противном случае, добавляем переход $\mathbf{(v_0,x_0,y_0,v_1)}$ в **M(R)**. Далее подаем стимул $\mathbf{x_1}$ допустимый по **M** в постсостоянии $\mathbf{v_1}$ и так далее. Реально тест будет строить только подавтомат **M(R)**.

При таком тестировании предполагается, что для каждого состояния **v**, достижимого по **M** из начального состояния $\mathbf{v_0}$, все стимулы, допустимые по **M**, допустимы и по **R** (обратное не требуется), если, конечно, «по дороге» от $\mathbf{v_0}$ до **v** не обнаружено ошибок в реализации. Будем называть это *гипотезой о допустимости* для тестирования с открытым состоянием.

Спецификация автомата, вообще говоря, может определять несколько спецификационных переходов из данного пресостояния по данному стимулу с получением одной и той же реакции; такие переходы различаются только своим постсостоянием. При тестировании с открытым состоянием нам известно единственное текущее постсостояние. Если состояние скрыто, будем требовать единственности в модели такого постсостояния, то есть, уравнение постусловия **POST(v,x,y,v`)=true** должно иметь не более одного решения относительно постсостояния **v`**. Будем говорить, что такая спецификация и описываемый ею модельный автомат *слабо-детерминированы*. В [11] это называется *наблюдаемым* недетерминизмом.

Предположим, что уравнение постусловия не только имеет не более одного решения, но и может быть вычислено. Например, постусловие имеет вид **“ReactionChecking(v,x,y) & v`=Poststate(v,x,y)”**, где **ReactionChecking** – предикат, определяющий правильность реакции, а **Poststate** – эксплицитная функция, вычисляющая постсостояние для правильной реакции. Тогда при определении перехода в автомате **M(R)**, соответствующего переходу (v_i,x_i,y_i,v_{i+1}) в **R(M)**, мы вместо неизвестного нам реализационного постсостояния v_{i+1} вычисляем модельное постсостояние v^*_{i+1} и добавляем в **M(R)** переход $(v^*_i,x_i,y_i,v^*_{i+1})$; здесь v^*_i – модельное состояние, вычисленное на предыдущем шаге, а в начале предполагается $v^*_0=v_0$.

Заметим, что при построении **M(R)** единственность и вычислимость постсостояния нам требуется не во всей модели **M**, а только в той ее части, которая используется при таком построении. Модель может не быть слабо-детерминированной в тех частях, куда мы можем попасть только по реакциям, отсутствующим в реализации и, следовательно, в **M(R)**. В дальнейшем, говоря о слабо-детерминированности, мы будем иметь в виду слабо-детерминированность **M(R)**.

При тестировании со скрытым состоянием мы также используем *гипотезу о допустимости*, но только речь идет не об одинаковых, а о *соответствующих* состояниях реализации – **v** и модели – v^*, для которых мы требуем, чтобы любой стимул, допустимый по **M** в v^*, допустим по **R** в **v**. Соответствие здесь понимается в следующем смысле: **v** и v^* достижимы из начального состояния v_0, соответственно, в **R** и в **M**, по одной и той же последовательности стимулов и реакций.

Гипотеза о допустимости, кажущаяся на первый взгляд немотивированной, на практике оказывается вполне естественной. Она означает, что допустимость стимулов в каждый момент времени однозначно определяется историей (последовательностью подаваемых стимулов и получаемых реакций), что вполне естественно при работе пользователя с программной системой, моделируемой автоматом. Естественно, предполагается, что ошибки, возможные в реализации, могут быть обнаружены (по получаемым реакциям) до того, как они приведут к нарушению «гипотезы о допустимости».

Мы описали построение подавтомата **M(R)** в процессе тестирования, но возникает вопрос: когда такое построение можно считать законченным? Одно условие очевидно: в каждом модельном состоянии v^*, которое попало в **M(R)**, мы должны попробовать все допустимые в нем стимулы, то есть, для каждого стимула **x**, допустимого по **M** в v^*, автомат **M(R)** содержит хотя бы один переход вида $(v^*,x,y,v^{*}``)$. Будем говорить, что в этом случае совершен *обход по стимулам* графа **M(R)**. Для детерминированного случая это означает, что **M(R)** построен и совершен его обход. В недетерминированном случае это условие необходимо, но не достаточно, поскольку таких переходов может быть несколько, и без дополнительных предположений у нас никогда не может быть гарантии, что все такие переходы уже попали в **M(R)**. Поэтому будем считать, что мы построили некоторое приближение $M`(R)\subseteq M(R)$.

При некоторых естественных предположениях о гарантированной достижимости одного состояния из другого можно считать, что **M`(R)** является суграфом **M(R)**, то есть, содержит все его состояния. В **M`(R)** не хватает лишь некоторых переходов $(v^*,x,y,v^{*}``)$, причем **M`(R)** содержит как $v^*$ и $v^{*}``$, так и хотя бы один переход из v^* по стимулу **x**. На второй фазе тестирования какие-то из этих переходов могут проходиться и добавляться в **M`(R)**.

Тестовая система на первой фазе содержит:

- *Алгоритм обхода по стимулам*.
- *Итератор* стимулов, определяемый предусловием спецификации, которое задает допустимость каждого стимула **x** в каждом состоянии **v** автомата: **PRE(v,x)=true**.
- *Медиатор*, предназначенный для подачи стимула на тестируемый автомат и получения реакции.
- *Оракул*, проверяющий правильность перехода и определяемый постусловием спецификации, которое задает допустимость полученных реакции **y** и постсостояния **v`** при переходе из пресостояния **v** по стимулу **x**: **POST(v,x,y,v`)=true**.
- *Определитель постсостояния*:
    a. для открытого состояния – операция *status message*,
    b. для скрытого состояния – эксплицитная функция **Poststate**.

В заключении кратко укажем еще на две проблемы, обычно возникающие при тестировании автоматов на практике: нетождественное медиаторное соответствие и факторизованное тестирование.

До сих пор мы предполагали, что алфавиты стимулов, реакций и состояний одинаковы в модели и реализации, а медиатор и функция **Poststate**, фактически, осуществляют тождественные преобразования. Однако, на практике такое встречается редко. С одной стороны, это объясняется техническими причинами, например, спецификации написаны на языке спецификаций, отличном от языка программирования реализации и имеющем другую типовую структуру. Но более важно то, что спецификация – это всегда некоторая абстракция и предназначена служить моделью для нескольких, подчас весьма сильно отличающихся одна от другой реализаций. В общем случае медиатор осуществляет нетождественные **down**-преобразование модельных стимулов в реализационные и **up**-преобразование реализационных реакций в модельные. При тестировании с открытым состоянием функция **Poststate** осуществляет **up**-преобразование реализационных состояний в модельные, а при тестировании со скрытым состоянием такое соответствие состояний существует неявно. В любом случае **up**-преобразования могут быть неинъективными, что приводит к некоторому «огрублению» тестирования, соответствующему уровню абстракции модели. Тем не менее, при открытом состоянии преобразования стимулов и реакций могут зависеть от реализационного состояния и, вообще говоря, даже от предыстории взаимодействия с реализационным автоматом.

«Огрубление» тестирования часто делается вполне сознательно как факторизация автомата, когда число состояний и стимулов (даже в подавтоматах **R(M)** и **M(R)**) слишком велико [13]. Факторизация проводится по заданной эквивалентности состояний и/или стимулов, а в общем случае – переходов, и требуется проверять только фактор-переходы, для чего может быть выбран любой переход из соответствующего класса эквивалентности. Заметим, что если считать эквивалентными переходы из данного состояния по данному стимулу, то получаемый в конце первой фазы тестирования подавтомат **M`(R)** можно рассматривать как такой фактор-автомат (если мы не вводим независимой эквивалентности состояний). В общем случае, кроме двух уровней реализации **R** и спецификационной модели **M**, мы получаем третий уровень фактор-модели **F(M)**. Тест работает уже не на уровне **M**, а на уровне **F(M)**, которую можно назвать тестовой моделью. Соответственно, на первой фазе будет строиться не подавтомат **M(R)** спецификационной модели, а подавтомат **F(M)(R)** тестовой модели, что совпадает с факторизацией **M(R)**, то есть, **F(M(R))**.

Факторизацию можно рассматривать как частный случай общего медиаторного соответствия между уровнем модели и фактор-модели. Обобщая, можно рассматривать многоуровневые системы описания автоматов, где каждый уровень связан медиаторными соответствиями с соседними уровнями, нижний уровень – это реализация, а верхний – это тестовая модель, то есть, модель, по которой непосредственно происходит тестирование реализации.

Проблемы второй фазы тестирования, нетождественных медиаторных соответствий и тестирование для многоуровневых систем мы здесь рассматривать не будем. В остальной части статьи речь будет идти только о центральном компоненте первой фазы тестирования – неизбыточных алгоритмах обхода по стимулам недетерминированных графов. Напомним, что неизбыточными мы называем алгоритмы, получающие информацию о графе в процессе движения по нему.

3. Понятия графов и алгоритмов

Ориентированным графом (далее просто *графом*) **G** будем называть совокупность трёх объектов: **VG** – множество вершин, **XG** – множество стимулов, **EG⊆VG×XG×VG** – множество дуг.

Стимул **x** *допустим* в вершине **a**, если в графе существует дуга **(a,x,b)**. Вершину **a** будем называть *началом* дуги, вершину **b** – *концом* дуги, стимул **x** – *раскраской* дуги. Если стимул дуги несущественен, мы будем также вместо **(a,x,b)** писать просто **(a,b)**. *Δ-дугой* **(a,x)** будем называть множество дуг графа, начинающихся в вершине **a** (начало Δ-дуги) и помеченных допустимым в **a** стимулом **x** (раскраска Δ-дуги): **(a,x)={(a,x,b)∈EG}**.

Замечание: При тестировании граф рассматривается как граф состояний автомата. Однако, в алгоритмах обхода не используется раскраска дуг реакциями, поскольку для алгоритма достаточно при проходе любой дуги уметь определять конец дуги (постсостояние). Это делает определитель постсостояния, который мы считаем внешним для алгоритма обхода. В связи с этим мы не различаем кратные дуги (они могут различаться только реакциями).

Граф называется конечным, если множества вершин и дуг конечны. Число вершин, дуг и Δ-дуг конечного графа обозначим, соответственно, **n**, **k** и **m**.

Граф называется *детерминированным*, если конец дуги однозначно определяется ее началом и допустимым в нем стимулом: для дуг **(a,x,b)** и **(a`,x`,b`)** из **a=a`** и **x=x`** следует **b=b`**. Все Δ-дуги такого графа являются синглетонами, то есть, состоят из одной дуги. В недетерминированном графе Δ-дуга может содержать несколько дуг, отличающихся своими концами.

Дуги **(a,x,b)** и **(a`,x`,b`)** называются *смежными*, если конец первой дуги совпадает с началом второй дуги: **b=a`**. *Маршрутом* **P** длины **n** в графе **G** называется последовательность **n** смежных дуг: для **i=1..n-1** дуга **P[i]** смежна с дугой **P[i+1]**. Начало **a** первой дуги маршрута будем называть его *началом*, конец **b** последней дуги маршрута – его *концом*, сам маршрут – **[a,b]**-маршрутом. Пустой последовательности дуг соответствует маршрут нулевой длины, начало и конец которого совпадают. Маршрут будем называть *обходом*, если он содержит все дуги графа, и *обходом по стимулам*, если он содержит крайней мере одну дугу из каждой Δ-дуги графа.

Δ-маршрутом будем называть множество маршрутов **D**, начинающихся в одной вершине, которое «ветвится» в каждой проходимой Δ-дуге. Формально: для каждого маршрута **P∈D** и каждого **i** меньшего длины **P** множество **i+1**-ых дуг маршрутов из **D**, совпадающих с **P** по первым **i** дугам, образует Δ-дугу, которой принадлежит **i+1**-дуга **P**: для **P[i+1]=(a,x,b)** **{Q[i+1]|Q∈D&Q[1..i]=P[1..i]}=(a,x)**. Начало **a** маршрутов Δ-маршрута **D** будем называть *началом* Δ-маршрута. Если все маршруты Δ-маршрута **D** заканчиваются в одной вершине **b**, будем называть ее *концом* Δ-маршрута, а сам Δ-маршрут – **[a,b]**-Δ-маршрутом. *Длиной* Δ-маршрута **D** будем называть максимальную длину его маршрутов. *Δ-обходом* будем называть Δ-маршрут, все маршруты которого являются обходами по стимулам.

Наше понятие Δ-маршрута близко понятию адаптивной тестовой последовательности или тестовому дереву, используемому для описания выбора стимулов в зависимости от выполнения предыдущих переходов [9]. Для детерминированного графа понятия маршрута и Δ-маршрута совпадают; более строго, каждый Δ-маршрут является синглетоном. Соответственно, понятие обхода совпадает с понятиями обхода по стимулам и Δ-обхода.

Алгоритмом движения по графу будем называть алгоритм, который в процессе своей работы строит маршрут в графе. Формально такой алгоритм можно определить как специальный вид машины с абстрактным состоянием (машина Гуревича, ASM – Abstract State Machine [15]), в котором внешние операции частично специфицированы заданием графа, на котором происходит работа алгоритма, и текущей вершины в нем. Для наших целей достаточно указать, что алгоритму предоставляются две специальные внешние операции **status()**, возвращающая идентификатор текущей вершины, и **call(x)**, которая осуществляет переход из текущей вершины **a** по дуге **(a,x,b)**, выбираемой неспецифицированным образом из Δ-дуги **(a,x)**. Для детерминированного графа такая дуга **(a,x,b)** единственная (единственна вершина **b**). Предусловием операции **call(x)** является допустимость стимула **x** в текущей вершине **a**. Маршрут строится алгоритмом как последовательность дуг, проходимых последовательными вызовами операции **call**. Следует отметить, что никакая внешняя операция (не говоря уже о внутренней) не меняет сам граф, и единственной модифицирующей операцией, которая может изменить текущую вершину, является операция **call**.

Неизбыточным алгоритмом будем называть алгоритм движения по графу, который зависит только от пройденной части графа и допустимости стимулов в текущей вершине. Допустимость стимулов алгоритм может определить с помощью специальной внешней операции **next()**, которая возвращает стимул, неспецифицированным образом выбираемый среди не выбранных ранее стимулов, допустимых в текущей вершине, осуществляя тем самым итерацию стимулов в вершине. Если все стимулы, допустимые в текущей вершине, уже выбирались, будем считать, что **next()** возвращает «пустой» символ **ε**.

Свободным алгоритмом будем называть неизбыточный алгоритм, который узнает о допустимости стимула, еще не опробованного в текущей вершине **a**, не заранее, а одновременно с проходом по дуге, раскрашенной этим стимулом. Иначе говоря, свободный алгоритм осуществляет первичный проход по любой еще не пройденной Δ-дуге с началом в текущей вершине, используя совмещенную внешнюю операцию **nextcall()**: **x=next(); if x≠ε then call(x); return x else return ε end**. Эта операция неспецифицированным образом выбирает еще не опробованный в текущей вершине **a** стимул **x** и проходит по дуге **(a,x,b)**, неспецифицированным образом выбираемой из Δ-дуги **(a,x)**. Если все стимулы уже опробованы в текущей вершине, возвращается пустой

символ **ε.** Для вторичного прохода по Δ-дуге **(a,x)**, по-прежнему, используется операция **call(x)** в тот момент, когда текущей вершиной является вершина **a**.

Алгоритм предназначен для решения той или иной задачи; в данной статье такой задачей является обход графа по стимулам. Работа алгоритма, вообще говоря, зависит от выполнения внешних операций **call** и **next**. В недетерминированном графе Δ-дуга содержит, вообще говоря, несколько дуг, различающихся своими концами, и выполнение операции **call(x)** неоднозначно определяется текущей вершиной и стимулом **x**. Поэтому обход по стимулам, который совершает алгоритм, вообще говоря, не является обходом графа и зависит от выполнения операции **call**. Мы можем говорить лишь о **call**-независимом множестве таких маршрутов, то есть, множестве всех маршрутов, которые может совершать алгоритм при зафиксированном выполнении всех внешних операций, кроме **call**, и всех возможных выполнениях операции **call**. Это множество, очевидно, является Δ-маршрутом в графе; каждый маршрут из этого Δ-маршрута соответствует некоторому варианту выполнения операции **call**. Если этот Δ-маршрут является Δ-обходом, то будем говорить, что алгоритм совершает Δ-обход графа с данной начальной вершиной. Будем говорить, что алгоритм совершает *гарантированный* Δ-обход графа с данной начальной вершиной, если он совершает обход по стимулам при любом допустимом выполнении всех внешних операций (а не только **call**); для неизбыточных алгоритмов – независимо от итерации стимулов в вершинах (**next**).

Нас будут интересовать только такие алгоритмы, которые останавливаются через конечное число шагов. В момент остановки алгоритм может сообщить, выполнен обход по стимулам или нет. Возможен также случай, когда построенный маршрут является обходом по стимулам, но алгоритм об этом «не знает». Противоположный случай, когда обход по стимулам не выполнен и алгоритм «знает», что в данном графе нет Δ-обхода. Подобную информацию, сообщаемую алгоритмом в момент остановки, будем называть *вердиктом* алгоритма. Мы будем говорить, что вердикт *достоверен*, если сообщаемая в нем информация соответствует действительности.

4. Δ-обход графов

4.1. Сильно-Δ-связные графы

Лемма 4.1: Если каждый маршрут Δ-маршрута **D**, начинающегося в вершине **a**, проходит через вершину **b**, то существует **[a,b]**-Δ-маршрут.

Искомый **[a,b]**-Δ-маршрут строится как множество начальных отрезков маршрутов из **D**, заканчивающихся первым вхождением вершины **b**. □

Непустое собственное подмножество вершин **U** графа будем называть Δ-*изолированным множеством*, если каждая Δ-дуга, имеющая начало в **U**, содержит дугу, имеющую конец в **U**. Если вершин **a** принадлежит Δ-изолированному множеству **U**, а вершина **b** не принадлежит **U**, то такое множество **U** будем называть **[a,b]**-Δ-изолированным.

Лемма 4.2: Если в графе существует **[a,b]**-Δ-маршрут **D**, то не существует **[a,b]**-Δ-изолированного множества вершин.

Допустим обратное: существует такое множество **U**. Каждый маршрут $\mathbf{P}\in\mathbf{D}$ ведет из $\mathbf{a}\in\mathbf{U}$ в $\mathbf{b}\notin\mathbf{U}$. Следовательно, в **P** существует дуга, которая ведет из **U** во вне его. Пусть **i** - индекс первой такой дуги **P[i]=(c,x,d)**, начало которой $\mathbf{c}\in\mathbf{U}$, а конец $\mathbf{d}\notin\mathbf{U}$. Рассмотрим

такой маршрут **P**, в котором индекс **i** максимальный. Поскольку **U** Δ-изолированное множество, в Δ-дуге **(c,x)** должна существовать дуга **(c,x,d`)** такая, что **d`∈U**. Но тогда, поскольку **D** Δ-маршрут, должен существовать маршрут **P`∈D** такой, что **P`[1,i-1]=P[1,i-1]** и **P`[i]=(c,x,d`)**. Очевидно, что в маршруте **P`** первая дуга, выходящая за пределы множества **U** имеет индекс больше, чем максимальный индекс **i**. Мы пришли к противоречию и, тем самым, Лемма доказана. □

Δ-*подграфом* будем называть такой подграф, который каждую Δ-дугу графа содержит или не содержит целиком (все ее дуги). *Δ-путем* будем называть Δ-маршрут, в котором все маршруты являются путями. Очевидно, длина Δ-пути не превосходит **n-1**. Граф называется ациклическим, если в нем нет циклических маршрутов; источником называется вершина, в которую не входят дуги; стоком называется вершина, из которой не выходят дуги.

Лемма 4.3: Если в графе **G** не существует **[a,b]**-Δ-изолированного множества вершин, то существует **[a,b]**-Δ-путь.

Сначала приведем алгоритм построения ациклического Δ-подграфа, в котором вершина **a** является источником (быть может, не единственным), а вершина **b** – единственным стоком. В начале каждого шага алгоритма мы будем иметь ациклический Δ-подграф **H** с единственным стоком **b**. В самом начале **H** состоит из одной вершины **b** и не имеет дуг. Шаг алгоритма заключается в следующем: если **a** не принадлежит **VH** и существует Δ-дуга **(c,x)**, начало которой **c∉VH**, а концы всех ее дуг принадлежат **VH**, то добавляем эту Δ-дугу в подграф **H** (ее начало в **VH** и все ее дуги в **EH**). В противном случае, алгоритм останавливается.

Заметим, что если **a∉VH**, то такая Δ-дуга **(c,x)** существует, поскольку в противном случае множество вершин **VG\VH** было бы **[a,b]**-Δ-изолированным. Следовательно, учитывая конечность числа Δ-дуг в графе, алгоритм через конечное число шагов остановится сразу после того, как вершина **a** попадет в **VH** и, очевидно, станет его источником. Поскольку на каждом шаге мы добавляем в **H** Δ-дугу целиком, **H** остается Δ-подграфом. Поскольку начало добавляемой Δ-дуги не принадлежало ранее **VH**, **H** остается ациклическим графом.

Искомый **[a,b]**-Δ-путь строится как множество маршрутов в Δ-подграфе **H**, начинающихся в **a** и заканчивающихся в **b**. □

Из Лемм 4.1-4.3 и определения **[a,b]**-Δ-пути непосредственно следует следующая теорема:

Теорема 4.1: Для вершин **a** и **b** в графе следующие утверждения эквивалентны: 1) существует Δ-маршрут, начинающийся в **a**, каждый маршрут которого проходит через **b**, 2) существует **[a,b]**-Δ-маршрут, 3) не существует **[a,b]**-Δ-изолированного множества вершин, 4) существует **[a,b]**-Δ-путь. □

Будем говорить, что из вершины **a** Δ-*достижима* вершина **b**, если верно любое из эквивалентных утверждений Теоремы 4.1. Граф называется *сильно*-Δ-*связным*, если из каждой его вершины Δ-достижима каждая его вершина.

Теорема 4.2: 1) Для любого сильно-Δ-связного графа и любой пары его вершин **a** и **b** всегда существует **[a,b]**-Δ-обход с длиной **O(nm)**. 2) Для любых **n** и **m** существует сильно-

Δ-связный граф с числом вершин **n** и числом Δ-дуг **m`≥m**, в котором любой Δ-обход имеет длину **Ω(nm`)**.

1) Произвольным образом линейно упорядочим все Δ-дуги графа так, чтобы первая Δ-дуга начиналась в вершине v_1=**a**: $(v_1,x_1),(v_1,x_1),\ldots,(v_m,x_m)$. Обозначим v_{m+1}=**b**. Из конца $v_i`$ любой дуги из **i**-ой Δ-дуги $(v_i,x_i,v_i`)\in(v_i,x_i)$ в сильно-Δ-связном графе всегда существует $[v_i`,v_{i+1}]$-Δ-путь $D_i(v_i`)$ в начало v_{i+1} следующей **i+1**-ой Δ-дуги (для **i=m** в вершину v_{m+1}=**b**). Объединение всех таких Δ-путей обозначим $D_i=\{D_i(v_i`)|v_i`\in(v_i,x_i)\}$. Искомый Δ-обход представляет собой множество всех возможных конкатенаций дуг и путей: $D=(v_1,v_1`)\wedge D_1\wedge\ldots\wedge(v_i,v_i`)\wedge D_i\wedge\ldots\wedge(v_m,v_m`)\wedge D_m$. В **D** каждый маршрут проходит хотя бы одной дуге из каждой Δ-дуги, начинается в **a**, заканчивается в **b** и поэтому является **[a,b]**-обходом по стимулам. Каждый такой обход по стимулам состоит из **m** дуг (по одной из каждой Δ-дуги) и **m** соединяющих путей (входящих в соединяющие Δ-пути), поэтому он имеет длину не более **m+m(n-1)=O(nm)**.

2) Поскольку в детерминированном графе Δ-дуга совпадает с дугой, а Δ-обход – с обходом, примером может служить детерминированный граф на рис.1, где **p=k`/n** – полустепень выхода каждой вершины.

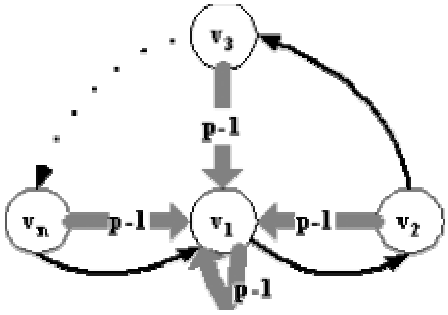

Рис.1

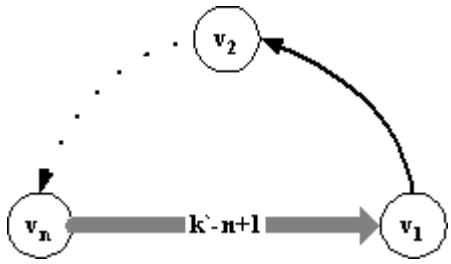

Рис.2

Обход этого графа можно представить в виде конкатенации маршрутов $P_1,\ldots,P_t$, причем каждый из $P_1,\ldots,P_{t-1}$ заканчивается дугой (v_i,v_1), где **i>1**, а каждый из $P_2,\ldots,P_{t-1}$ начинается в v_1. Каждый маршрут из $P_2,\ldots,P_{t-1}$, заканчивающийся дугой (v_i,v_1) имеет длину **i**, а их число равно **p-1**. Поэтому, считая, что маршрут P_1 заканчивается дугой (v_j,v_1) и имеет длину не менее 1, и не учитывая последний маршрут P_t, получаем нижнюю оценку длины обхода
$(p-1)+2(p-1)+\ldots+n(p-1)-j+1=(p-1)n(n-1)/2-j+1\geq(p-1)n(n-1)/2-n+1=L$. Нам достаточно, чтобы для некоторой константы **C>0** при любом **n>0** выполнялось $L\geq Cpn^2=Ck`n$. Легко показать, например, что это так для **C=1/3** и **p≥3**. Тем самым, мы определяем **k` = max{k,3n}**.

Заметим, что в примере на рис.1 полустепени выхода у всех вершин одинаковые. Это сделано для того, чтобы не накладывать на число стимулов ограничений снизу, кроме необходимого **k`/n**. Без этого требования пример можно упростить, заменив все дуги, ведущие из $v_i$ в $v_1$ на дуги, ведущие из $v_n$ в $v_1$, что требует **k`–n+1** стимулов (рис.2).

В примерах на рис.1,2 есть жесткая зависимость между числом дуг и Δ-дуг **k`=m`**. Если мы хотим, например, чтобы число дуг было в **t** раз больше числа Δ-дуг, можно для каждой дуги этого графа добавить еще одну Δ-дугу с тем же началом, состоящую из **2t-1** дуг, ведущих в любые вершины. Число вершин не изменится, число дуг **k``=k`+k`(2t-1)=2tk`**, а число Δ-дуг **m``=2k`**. Очевидно, что такой граф сильно-Δ-связен и длина любого его Δ-обхода **Ω(nk`)= Ω(nm``)**. □

4.2. Графы Δ-достижимости

Поскольку взаимная Δ-достижимость вершин – это отношение эквивалентности, в общем случае граф **G** разбивается на компоненты сильной Δ-связности, на множестве которых Δ-достижимость является отношением частичного порядка. Компонент является подграфом графа **G**, множество вершин которого – это класс эквивалентности, а дуги – все дуги графа **G**, начало и конец которых принадлежат этому классу. Компонент, которому принадлежит вершина **a**, будем обозначать **K(a)**. Будем говорить, что дуга **(a,x,b)** ведет *вперед*, если из **K(b)** Δ-недостижим **K(a)**. Δ-дугу будем называть *связующей*, если концы всех ее дуг принадлежат одному компоненту **B**, отличному от компонента **A** начала Δ-дуги; будем говорить, что связующая Δ-дуга ведет из **A** в **B**.

Для графа **G** его *фактор-графом* по отношению взаимной Δ-достижимости будем называть граф **F(G)**, вершинами которого являются компоненты сильной Δ-связности графа **G**, а дуга **(A,x,B)**, где **A≠B** – компоненты графа **G**, – это связующая Δ-дуга графа **G**, ведущая из **A** в **B** и раскрашенная стимулом **x**. Фактор-граф, очевидно, является детерминированным ациклическим графом.

Δ-*достижимым графом* будем называть граф, все вершины которого Δ-достижимы из выделенной начальной вершины. В силу Леммы 4.1, Δ-обход возможен только в Δ-достижимом графе, поэтому в дальнейшем, специально не оговаривая это, мы будем рассматривать только Δ-достижимые графы.

Теорема 4.3: Для того, чтобы граф **G** был Δ-достижимым графом с начальной вершиной $\mathbf{v_0}$, необходимо и достаточно, чтобы его фактор-граф **F(G)** по отношению Δ-достижимости был ациклическим графом с одним источником $\mathbf{K(v_0)}$.

Необходимость. В любой компонент $\mathbf{K\neq K(v_0)}$ входит хотя бы одна связующая Δ-дуга **(v,x)**, так как в противном случае, множество **VG\VK** было бы Δ-изолированным в **G** и никакая вершина из **VK** не была бы Δ-достижима из $\mathbf{v_0}$. Следовательно, в каждую фактор-вершину фактор-графа, кроме $\mathbf{K(v_0)}$, входит хотя бы одна фактор-дуга, и **F(G)**, как ациклический граф имеет только один источник $\mathbf{K(v_0)}$.

Достаточность. Для любой вершины **v** рассмотрим в **F(G)** путь **F** длиной **t** из $\mathbf{K(v_0)}$ в **K(v)**. Его **i**-ая фактор-дуга – это связующая Δ-дуга графа $\mathbf{G\ (v_i,x_i)}$, где **i=1..t**. Рассматривая все ее дуги $\mathbf{(v_i,x_i,v_i`)\in(v_i,x_i)}$ и обозначая $\mathbf{v_{t+1}=v}$, получаем, что каждая пара вершин $\mathbf{v_i`}$, $\mathbf{v_{i+1}}$, где **i=1..t**, принадлежит одному компоненту; поэтому существует $\mathbf{[v_i`,v_{i+1}]}$-Δ- маршрут $\mathbf{D(v_i`)}$. Объединение всех таких Δ-путей обозначим $\mathbf{D_i=\{D_i(v_i`)|v_i`\in(v_i,x_i)\}}$. Через $\mathbf{D_0}$ обозначим $\mathbf{[v_0,v_1]}$-Δ-маршрут в компоненте $\mathbf{K(v_0)}$. Искомый $\mathbf{[v_0,v]}$-Δ-маршрут строится как множество всех возможных конкатенаций $\mathbf{D_0\hat{}(v_1,x_1)\hat{}\ D_1\hat{}\ldots\hat{}(v_t,x_t)\hat{}D_t}$. □

4.3. Графы 1-го рода

Графом 1-*го рода* будем называть граф с линейным порядком Δ-достижимости компонентов, в котором для каждого непоследнего **i**-го компонента все дуги, выходящие из него и ведущие *вперед* (в компоненты с большим номером), образуют одну связующую Δ-дугу, ведущую в следующий **i+1**-ый компонент. Заметим, что некоторые дуги (но не Δ-дуги!) могут вести *назад*, в компоненты с меньшими номерами. По умолчанию, начальная вершина принадлежит первому компоненту (рис.3). Для детерминированного случая наше определение графа 1-го рода совпадает с тем, которое дано в [18].

Рис.3

Пройденным графом Δ-маршрута **D** будем называть подграф **GD**, состоящий из всех дуг всех маршрутов этого Δ-маршрута и инцидентных им вершин. Пройденный граф, очевидно, является Δ-подграфом. Заметим, что **D** может не быть Δ-обходом **GD**, в отличии от детерминированного случая, где маршрут всегда является обходом своего пройденного графа [18].

Теорема 4.4: 1) Если Δ-маршрут **D** является Δ-обходом своего пройденного графа **GD**, то **GD** – граф 1-го рода и начало $\mathbf{v_0}$ Δ-маршрута принадлежит его первому компоненту. 2) Для существования **[a,b]**-Δ-обхода **D** необходимо и достаточно, чтобы граф был графом 1-го рода, в котором вершины **a** и **b** принадлежат, соответственно, первому и последнему компонентам; минимальная длина Δ-обхода равна **O(nm)**. 3) Для любых возможных **n** и **m** существует граф 1-го рода с **n** вершинами и $\mathbf{m`\geq m}$ Δ-дугами, любой Δ-обход которого имеет длину $\mathbf{\Omega(nm`)}$. 4) Δ-обход из любой начальной вершины существует для и только для сильно-Δ-связных графов.

1) Для любых двух вершин **b** и **c** любой машрут из Δ-обхода проходит через обе эти вершины. Пусть в маршруте $\mathbf{P\in D}$ первое вхождение **b** предшествует первому вхождению **c**, и пусть **P(b)** - начальный отрезок **P**, заканчивающийся первым вхождением **b**. Все маршруты $\mathbf{Q\in D}$, продолжающие **P(b)**, в своих продолжениях **Q\P(b)** должны проходить через **c**. Множество этих продолжений, очевидно образует Δ-маршрут в пройденном графе, и, по Лемме 4.1, в пройденном графе существует **[b,c]**-Δ-маршрут. Таким образом, для любой пары вершин пройденного графа одна из из них Δ-достижима из другой в пройденном графе, поэтому пройденный граф имеет линейный порядок Δ-достижимости компонентов. Поскольку в каждом маршруте из D начальная вершина $\mathbf{v_0}$ проходится раньше любой другой вершины, из $\mathbf{v_0}$ Δ-достижима любая другая пройденная вершина и, следовательно, $\mathbf{v_0}$ принадлежит первому компоненту.

Нам осталось показать, что любая дуга **(b,x,c)**, ведущая в пройденном графе из **i**-го компонента *вперед* (в компонент с номером больше **i**), принадлежит связующей Δ-дуге $\mathbf{(a_i,x_i)}$. Допустим обратное $\mathbf{(b,x)\neq(a_i,x_i)}$. Любой маршрут $\mathbf{P\in D}$ должен проходить через обе эти Δ-дуги. Если **P** сначала проходит через Δ-дугу $\mathbf{(a_i,x_i)}$, $\mathbf{P[j]\in(a_i,x_i)}$, то он попадает в **i+1**-ый компонент, из которого **i**-ый компонент Δ-недостижим, и, следовательно, среди маршрутов **D**, продолжающих **P[1..j]**, найдется хотя бы один, не попадающий в **b**, и, тем самым, не проходящий через Δ-дугу **(b,x)**, чего быть не может, поскольку **D** - Δ-обход **GD**. Если **P** сначала проходит через Δ-дугу **(b,x)**, $\mathbf{P[j]\in(b,x)}$, то существует такой маршрут $\mathbf{P`\in D}$, который совпадает **P** по первым **j-1** дугам и проходит по дуге $\mathbf{P`[j]=(b,x,c)\in(b,x)}$. При этом он попадает в компонент, из которого **i**-ый компонент Δ-недостижим, и, следовательно, среди маршрутов **D**, продолжающих **P`[1..j]**, найдется хотя бы один, не попадающий в $\mathbf{a_i}$, и, тем самым, не проходящий через Δ-дугу $\mathbf{(a_i,x_i)}$, чего быть не может, поскольку **D** - Δ-обход **GD**.

2) Необходимость следует из 1), нужно только показать, что вершина **b** принадлежит последнему компоненту. Поскольку все маршруты **D** проходят через каждую вершину **c**, их продолжения после первого вхождения вершины **c**, образующие Δ-маршрут,

заканчиваются в **b**, и, тем самым, **b** Δ-достижима из каждой вершины **c**, то есть, принадлежит последнему компоненту.

Достаточность. Мы дадим алгоритм построения **[a,b]**-Δ-обхода. В начале каждого **i**-го шага мы имеем Δ-маршрут $\mathbf{D_i}$, в котором все маршруты $\mathbf{P \in D_i}$ являются начальными отрезками маршрутов будущего Δ-обхода. При этом каждый маршрут **P** начинается в вершине **a**, заканчивается в некоторой вершине **c** (у разных маршрутов могут быть разные концы), и проходит через все Δ-дуги всех компонентов, предшествующих компоненту конца маршрута **K(c)**. Шаг применяется к каждому маршруту из $\mathbf{D_i}$ и, если он не является **[a,b]**-обходом по стимулам, заменяет его множеством продолжающих его маршрутов так, что получается новый Δ-маршрут $\mathbf{D_{i+1}}$. Алгоритм заканчивается, если все маршруты являются **[a,b]**-обходами по стимулам. В самом начале имеем один маршрут нулевой длины с началом в **a**. Шаг алгоритма состоит в следующем:

1. Пусть маршрут **P** такой, что все Δ-дуги , выходящие из вершин компонента его конца **K(c)**, пройдены в **P**. Тогда, в силу условия шага, **P** является обходом по стимулам, но он может не заканчиваться в **b**. Строим **[c,b]**-Δ-путь $\mathbf{Q_b}$ и вместо маршрута **P** берем множество всех его конкатенаций с маршрутами из $\mathbf{Q_b}$. Условия шага, очевидно, остаются выполненными.
2. Пусть есть маршрут **P** такой, что из вершины $\mathbf{d \in VK(c)}$ выходит непройденная в **P** Δ-дуга **(d,x)**. Приоритет отдается несвязующим Δ-дугам, то есть, связующую Δ-дугу мы выбираем только в том случае, если все остальные Δ-дуги, выходящие из вершин **K(c)**, уже пройдены в **P**. Строим **[c,d]**-Δ-путь $\mathbf{Q_d}$ и вместо маршрута **P** берем множество всех его конкатенаций с маршрутами из $\mathbf{Q_d}$ и далее с дугами Δ-дуги **(d,x)**. Если все дуги Δ-дуги **(d,x)** заканчиваются в компоненте **K(c)** или Δ-дуга **(d,x)** связующая, то, очевидно, условия шага остаются выполненными.
3. В противном случае, некоторые дуги $\mathbf{(d,x,e) \in (d,x)}$ заканчиваются в компонентах $\mathbf{K(e) \neq K(c)}$. В графе 1_Δ-рода **K(e)** предшествует **K(c)**, то есть, из **e** Δ-достижимы все вершины **K(c)**. Тогда для каждого маршрута **P**, полученного в п.2, который заканчивается в такой вершине **e**, строим **[e,c]**-Δ-путь $\mathbf{Q_c}$, и вместо **P** берем все его конкатенации с маршрутами из $\mathbf{Q_c}$. Условия начала шага, очевидно, остаются выполненными.

Поскольку на каждом шаге каждый маршрут, не являющийся **[a,b]**-обходом по стимулам, заменяется продолжающими его маршрутами с увеличением числа пройденных Δ-дуг, алгоритм не более чем через **m** шагов закончится и **[a,b]**-Δ-обход будет построен. На каждом шаге мы удлиняем каждый маршрут не более чем на длину пути $\mathbf{Q_b}$ (п.1) или длину пути $\mathbf{Q_d}$ + одна дуга (п.2) плюс длину пути $\mathbf{Q_c}$ (п.3). Поскольку длина пути не превосходит **n-1**, длина каждого маршрута и, тем самым, длина построенного Δ-обхода не превосходит $\mathbf{m(2n-1)=O(nm)}$.

Утверждение 3) непосредственно следует из Теоремы 4.2 для сильно-Δ-связных графов.

Утверждение 4) непосредственно следует из 2) и определения сильно-Δ-связного графа как графа с одним компонентом сильно-Δ-связности. □

4.4. Покрытия Δ-достижимых графов

Будем говорить, что Δ-маршрут *покрывает* вершину (Δ--дугу) графа, если каждый его маршрут проходит через эту вершину (Δ-дугу). В этом смысле Δ-обход – это как раз такой Δ-маршрут, который покрывает все вершины и Δ-дуги графа. Множество Δ-маршрутов,

начинающихся в начальной вершине, будем называть Δ-*покрытием* графа, если каждая вершина и каждая Δ-дуга графа покрывается хотя бы одним из Δ-маршрутов множества. Длиной Δ-покрытия будем называть сумму длин его Δ-маршрутов.

Теорема 4.5: Δ-покрытие существует для и только для Δ-достижимых графов и его минимальная длина равна **O(nm)**; для любых возможных **n** и **m** существует граф Δ-достижимости с **n** вершинами и **m`≥m** Δ-дугами, любое Δ-покрытие которого имеет длину **Ω(nm`)**.

Утверждение теоремы о существовании Δ-покрытия непосредственно следует из определения Δ-достижимого графа **G** с начальной вершиной $\mathbf{v_0}$: для каждой вершины **v** имеется $\mathbf{[v_0,v]}$-Δ-маршрут **D(v)**, для каждой Δ-дуги **(v,x)** имеется проходящий через нее Δ-маршрут **D(v)^(v,x)** – множество всех конкатенаций маршрутов из **D(v)** и дуг из **(v,x)**.

Для оценки длины Δ-покрытия Δ-достижимого графа **G** рассмотрим фактор-граф **F(G**, который, по Теореме 4.3, является детерминированным ациклическим графом с одним источником $\mathbf{K(v_0)}$. Обозначим **t** – число компонентов, а $\mathbf{m_0}$ – число связующих Δ-дуг графа **G**. Выделим в фактор-графе остов (максимальное дерево), ориентированный от корня – источника; в нем **t-1** фактор-дуг, остальные $\mathbf{m_0-(t-1)}$ фактор-дуги - фактор-хорды. Для покрытия фактор-графа достаточно взять множество $\mathbb{F}$ следующих фактор-маршрутов: a) все фактор-маршруты, ведущие из корня во все листовые фактор-вершины, из которых не выходят фактор-хорды, и дополнительно b) все фактор-маршруты, ведущие из корня в начала всех фактор-хорд и далее проходящие по фактор-хорде. Число фактор-маршрутов a) не превосходит **t**, число фактор-маршрутов b) не превосходит $\mathbf{m_0-(t-1)}$; суммарное число фактор-маршрутов – не более $\mathbf{m_0+1}$.

Каждому фактор-маршруту $\mathbf{F\in\mathbb{F}}$ можно поставить в соответствие чередующуюся последовательность компонентов графа **G** (начала и концы фактор-дуг **F**) и связующих Δ-дуг графа **G** (фактор-дуг **F**): $\mathbf{K(v_1),(v_1,x_1),K(v_2),(v_2,x_2)\ldots(v_f,x_f),K(v_{f+1})}$, причем $\mathbf{K(v_1)=K(v_0)}$. В каждый компонент $\mathbf{K(v_i)}$, кроме первого, входит связующая Δ-дуга $\mathbf{(v_{i-1},x_{i-1})}$ из предыдущего компонента, и из каждого компонента $\mathbf{K(v_i)}$, кроме последнего, выходит одна связующая Δ-дуга $\mathbf{(v_i,x_i)}$, ведущая в следующий компонент. В каждом компоненте $\mathbf{K(v_i)}$, кроме первого и последнего, для каждой входящей в него дуги $\mathbf{(v_{i-1},x_{i-1},v_{i-1}`)\in(v_{i-1},x_{i-1})}$ выберем $\mathbf{[v_{i-1}`,v_i]}$-Δ-путь $\mathbf{D_i(v_{i-1}`)}$ из конца входящей дуги в начало выходящей Δ-дуги. В последнем компоненте такой Δ-путь $\mathbf{D_{f+1}(v_f`)}$ можно закончить в любой вершине, а в первом компоненте Δ-путь $\mathbf{D_0}$ будет начинаться в начальной вершине $\mathbf{v_0}$. Для **i>0** обозначим объединение этих Δ-путей $\mathbf{D_i=\cup\{D_i(v_i`)|(v_i,x_i,v_i`)\in(v_i,x_i)\}}$. Заменяя компонент на соответствующее множество путей, сопоставим фактор-маршруту **F** множество всех возможных чередующихся конкатенаций множеств путей и связующих Δ-дуг $\mathbf{D(F) = D_0\text{^}(v_1,x_1)\text{^}D_1\text{^}(v_2,x_2)\text{^}\ldots\text{^}(v_f,x_f)\text{^}D_{f+1}}$, которое, очевидно, является Δ-маршрутом с началом в вершине $\mathbf{v_0}$.

Теперь для каждого компонента **K** графа **G** выберем один фактор-маршрут **F**, который проходит через этот компонент $\mathbf{K=K(v_i)}$. В Δ-маршруте **D(F)** заменим каждый Δ-путь $\mathbf{D_i(v_{i-1}`)}$ в компоненте **K** на Δ-обход компонента с тем же началом и концом. Искомое Δ-покрытие теперь – это множество всех таких Δ-маршрутов $\mathbf{D=\{D(F)|F\in\mathbb{F}\}}$. Если бы все Δ-маршруты $\mathbf{D_i(v_{i-1}`)}$ оставались Δ-путями, каждый Δ-маршрут **D(F)** тоже был бы Δ-путем или Δ-путем, удлиненным на одну Δ-дугу (фактор-хорду), и, следовательно, его длина не превосходила бы **n**, а сумма длин – $\mathbf{(m_0+1)n}$. Поскольку каждый **i**-ый компонент графа **G**

Δ-обходится ровно для одного **F**, длина Δ-покрытия не превосходит **$(m_0+1)n+\Sigma\{O(n_i m_i)|i=1..\ t\}=O(nm)$**, где **$n_i$, m_i** – число вершин и Δ-дуг **i**-го компонента.

Оценка **Ω(nm`)** достигается на графе на рис.1,2 с начальной вершиной **v_1**. □

5. Алгоритмы Δ-обхода графов

5.1. Графы 2-го рода и свободные алгоритмы

Будем говорить, что Δ-дуга пройдена в маршруте, если пройдена хотя бы одна дуга этой Δ-дуги; вершину будем называть *полностью пройденной*, если пройдены все выходящие из вершины Δ-дуги.

Графом 2-*го рода* будем называть такой граф 1-го рода, в котором все компоненты, кроме, быть может, последнего, состоят из одной вершины и не содержат дуг, кроме дуг одной связующей Δ-дуги, ведущей в следующий компонент (рис.4). Поскольку все компоненты, кроме последнего, состоят из одной вершины, все связующие Δ-дуги, кроме последней, состоят из одной дуги. Для детерминированного случая наше определение графа 2-го рода совпадает с тем, которое дано в [18].

Рис.4

Теорема 5.1: Любой свободный алгоритм может совершить гарантированный Δ-обход только графа 2-го рода с начальной вершиной **v_0** из первого компонента и остановкой в вершине из последнего компонента.

Будем вести доказательство от противного. По Теореме 4.4, Δ-обход существует только для графа 1-го рода. Если граф не 2-го рода, то некоторый непоследний компонент графа либо состоит более, чем из одной вершины, либо из его единственной вершины выходит не только связующая Δ-дуга. В первом случае, в силу сильно-Δ-связности компонента, имеется Δ-маршрут из начала связующей Δ-дуги в другую вершину компонента, следовательно, из начала связующей Δ-дуги **(a,x)** выходит еще одна Δ-дуга **(a,x`)**, а во втором случае наличие таких двух Δ-дуг явно постулируется. Когда алгоритм впервые оказывается в вершине **a**, в ней еще ни один стимул не опробован, и поэтому свободный алгоритм должен применить операцию **nextcall**. Поскольку Δ-обход должен быть гарантированным, проходимый маршрут должен быть обходом по стимулам при любом выполнении операции **nextcall**. Предположим, что **nextcall** выбирает стимул **x**. В этом случае мы пройдем по дуге из связующей Δ-дуги **(a,x)** в вершину следующего компонента, из которой нет Δ-маршрута, ведущего в вершину **a**, и Δ-дуга **(a,x`)** останется непройденной. Мы пришли к противоречию. □

5.2. Свободный алгоритм с оптимальной сложностью

Теорема 5.2: Существует свободный алгоритм $\mathbb{B}_1$, который останавливается на любом графе и совершает гарантированный Δ-обход любого графа 2-го рода с начальной вершиной из первого компонента. Длина пройденного маршрута **O(nm)**. Время работы алгоритма зависит от операций сравнения, определенных для идентификаторов вершин:

если есть только сравнение на равенство, то $\mathbf{O(n^2m)}$; если есть также сравнение на больше/меньше, то $\mathbf{O(nm\log_2 n)}$. Требуемая память $\mathbf{O(nI+mX+m\log_2 m)}$ бит, где **I** и **X** – размер в битах, соответственно, идентификатора вершины и стимула.

Сначала введем некоторые понятия и обсудим общую идею алгоритма.

В детерминированном графе расстоянием от вершины **a** до вершины **b** называют длину минимального **[a,b]**-маршрута; соответственно, расстояние от вершины **a** до *множества* вершин **B** определяют как минимум расстояний от **a** до вершин $\mathbf{b\in B}$. Аналогично в недетерминированном графе можно ввести *Δ-расстояние* $\boldsymbol{\rho}$**(a)** *вершины* **a** до множества **B** как минимальную длину **[a,B]**-Δ-маршрута, начинающегося в **a** и кончающегося в **B**, то есть, концы всех его маршрутов принадлежат **B**. В сильно-Δ-связном графе **[a,B]**-Δ-маршрут всегда существует и, очевидно, минимальный из них является Δ-путем и поэтому $\boldsymbol{\rho(a)\le n-1}$. *Δ-расстояние Δ-дуги без петель* $\boldsymbol{\rho}$**(a,x)** определим как увеличенный на **1** максимум из Δ-расстояний концов ее дуг: $\boldsymbol{\rho(a,x)=\max\{\rho(c)+1|(a,x,c)\in(a,x)\}}$. Очевидно, что для $\mathbf{a\in B}$ $\boldsymbol{\rho(a)=0}$, а для вершины $\mathbf{a\notin B}$ $\boldsymbol{\rho(a)=\min\{\rho(a,x)\}}$, где **(a,x)** пробегает все Δ-дуги без петель, выходящие из вершины **a**.

Для свободного алгоритма, который в некоторый момент прошел маршрут **P**, *помеченной* вершиной будем называть вершину, в которой операция **nextcall** вернула пустой символ **ε**, то есть, такую вершину, о которой алгоритм «знает», что она полностью пройдена. В дальнейшем Δ-расстояния вершин и Δ-дуг всегда будем считать до множества *непомеченных* вершин.

Если бы алгоритму в каждый момент времени были известны Δ-расстояния вершин и Δ-дуг, он мог бы работать по следующей простой схеме:

1. Если все пройденные вершины помечены, алгоритм останавливается.
2. В противном случае, из непомеченных вершин двигаемся все время по непройденным ранее Δ-дугам с помощью операции **nextcall** до тех пор, пока не окажемся в помеченной вершине или **nextcall** вернет пустой символ **ε** и текущая вершина станет помеченной.
3. Если из помеченной вершины не выходят Δ-дуги, для которых определено Δ-расстояние, алгоритм останавливается.
4. В противном случае, двигаемся по выходящей Δ-дуге с наименьшим Δ-расстоянием до тех пор, пока не попадем в непомеченую вершину или в помеченную вершину, из которой не выходят такие Δ-дуги.

В п.4 алгоритм проходит путь длиной не более **n-1** и, если попадает в непомеченную вершину, в п.2 проходит новую Δ-дугу. Таким образом, до остановки в п.1 или п.3 алгоритм пройдет маршрут длиной **O(nm)**. Если граф сильно-Δ-связен, п.3 невозможен и алгоритм остановится в п.1, обойдя по стимулам весь граф. Для графа 2-го рода с начальной вершиной в первом компоненте алгоритм, очевидно, сначала двигается по связующим Δ-дугам до тех пор, пока не попадет в последний компонент, который затем обходится по стимулам. Однако в этом случае вершины непоследнего компонента останутся непомеченными и, хотя все Δ-дуги будут гарантированно пройдены, алгоритм остановится в п.3.

Такой алгоритм, очевидно, является избыточным. Его неизбыточную версию можно сделать, используя вместо Δ-расстояний, вычисляемых по всему графу, Δ-расстояния по *пройденному* графу. Такие Δ-расстояния нужно перевычислять каждый раз, когда

проходится новая дуга. Длина обхода по стимулам по-прежнему останется **O(nm)**, однако время работы алгоритма будет довольно большим.

Основная идея предлагаемого ниже алгоритма заключается в использовании локальной аппроксимации Δ-расстояния, которую мы будем называть *рангом* и обозначать **r(v)** – для вершины **v**, и **r(v,x)** – для Δ-дуги **(v,x)**. В процессе этой работы ранги могут корректироваться в сторону увеличения. Если ранг вершины становится равен или больше числа пройденных вершин, алгоритм останавливается. Это может произойти только в том случае, когда граф не сильно-Δ-связен.

Алгоритм поддерживает список пройденных вершин; при каждой пройденной вершине **v** хранится односторонний список рангов выходящих из этой вершины пройденных Δ-дуг без петель, упорядоченный по возрастанию рангов; при каждом таком ранге **r** хранится двусторонний список пройденных Δ-дуг без петель, выходящих из вершины **v** и имеющих ранг **r**.

Алгоритм использует следующие структуры данных:

- Список пройденных вершин. Описатель вершины **v** содержит:
  - Идентификатор вершины (возвращаемый операцией **status**).
  - Признак помеченной вершины (операция **nextcall** вернула пустой символ **ε**).
  - Ссылку на первый элемент списка рангов.
- Список рангов пройденных Δ-дуг без петель, выходящих из вершины **v**. Описатель ранга **r** содержит:
  - Значение ранга.
  - Ссылку по списку рангов.
  - Ссылку на первый элемент списка Δ-дуг.
- Список пройденных Δ-дуг без петель, выходящих из вершины **v** и имеющих ранг **r**. Описатель Δ-дуги **(v,x)** содержит:
  - Стимул Δ-дуги.
  - Ссылку «вперед» по списку Δ-дуг.
  - Ссылку «назад» по списку Δ-дуг.
- **N** - счетчик пройденных вершин.
- **L** - счетчик помеченных вершин.
- **v** - ссылка на описатель текущей вершины.

Ранг непомеченной вершины считается равным **0**, а ранг помеченной вершины равен минимальному из рангов выходящих Δ-дуг без петель, то есть, первого ранга в списке рангов выходящих из вершины пройденных Δ-дуг без петель. Если список рангов для помеченной вершины пуст, ранг вершины неопределен.

В начале работы алгоритма списки вершин и привязанные к ним списки рангов и Δ-дуг пусты, оба счетчика равны нулю, ссылка на текущую вершину **v** также пуста. Алгоритм с помощью операции **status** определяет идентификатор начальной вершины **v** и создает ее описатель: список рангов пуст, вершина непомечена. Счетчик пройденных вершин становится равным **1**.

В дальнейшем алгоритм выполняется как последовательность однотипных шагов. Каждый шаг содержит один вызов операции **nextcall** или **call**, при этом либо проходится одна дуга **(v,x,v`)**, либо текущая непомеченная вершина **v** становится помеченной (**nextcall** возвращает пустой символ **ε**). Ранги вершин и Δ-дуг, текущую вершину, а также значения

счетчиков в конце шага будем снабжать штрихом: **r`**, v`, **N`**, **L`**. Шаг алгоритма состоит в следующем:

1. Текущая вершина **v** непомечена. Вызываем операцию **nextcall**.
   A. **nextcall** возвращает пустой символ **ε**. Помечаем текущую вершину и увеличиваем счетчик помеченных вершин **L`=L+1**.
      a. Если после этого счетчики пройденных и помеченных вершин стали равны **N`=L`**, алгоритм останавливается (*стоп 1*).
      b. В противном случае **N`>L`**, анализируем ранг вершины **v**.
         I. Если список рангов Δ-дуг пуст, то есть, **r`(v)** неопределен, или **r`(v)≥N**, алгоритм останавливается (*стоп 2*).
         II. В противном случае, конец шага 1).
   B. **nextcall** совершает переход по непройденной Δ-дуге **(v,x)** и возвращает ее стимул **x**. С помощью операции **status** определяем идентификатор новой текущей вершины **v`**, то есть, конец пройденной дуги **(v,x,v`)**. По этому идентификатору ищем **v`** в списке описателей пройденных вершин.
      a. Если вершина **v`** не найдена (новая), создаем ее описатель: список рангов пуст, вершина непомечена. Увеличиваем счетчик пройденных вершин **N`=N+1**. Создаем описатель Δ-дуги **(v,x)** и помещаем ее в список Δ-дуг ранга **1**, выходящих из **v**. При необходимости создаем описатель ранга **1** для вершины **v**. Конец шага 2).
      b. Если вершина **v`** найдена (старая), то проверяем, не является ли пройденная дуга **(v,x,v`)** петлей.
         I. Петля: **v`=v**. Конец шага 3).
         II. Не петля: **v`≠v**. Ранг Δ-дуги **(v,x)** устанавливается **r`(v,x)=r(v`)+1**. Создаем описатель Δ-дуги и помещаем его в список Δ-дуг ранга **r`(v,x)**. При необходимости создаем описатель ранга **r`(v,x)** для вершины **v**. Конец шага 4).
2. Текущая вершина **v** помечена. Выбираем Δ-дугу **(v,x)**, выходящую из **v** и имеющую наименьший ранг. Для стимула **x** выбранной Δ-дуги выполняем **call(x)**. С помощью операции **status** определяем идентификатор новой текущей вершины **v`**, то есть, конец пройденной дуги **(v,x,v`)**. По этому идентификатору ищем **v`** в списке описателей пройденных вершин.
   A. Если вершина **v`** не найдена (новая), создаем ее описатель: список рангов пуст, вершина непомечена. Увеличиваем счетчик пройденных вершин **N`=N+1**. Конец шага 5).
   B. Если вершина **v`** найдена (старая), то проверяем, не является ли пройденная дуга **(v,x,v`)** петлей.
      a. Петля: **v`=v**. Описатель Δ-дуги **(v,x)** изымаем из списка Δ-дуг ранга **r(v,x)**. Если список Δ-дуг ранга **r(v,x)** стал пуст, удаляем описатель этого ранга из списка рангов и уничтожаем его. Корректируем ссылку из описателя вершины **v** на описатель следующего ранга; если список рангов стал пуст, эта ссылка также будет пустой. Анализируем ранг вершины **v**.
         I. Если список рангов Δ-дуг пуст, то есть, **r`(v)** неопределен, или **r`(v)≥N**, алгоритм останавливается (*стоп 2*).
         II. В противном случае, конец шага 6).
      b. Не петля: **v`≠v**. Сравниваем ранг **r(v`)** с рангом начала пройденной дуги **r(v)=r(v,x)**.
         I. **r(v)>r(v`)**. Ранги Δ-дуги **(v,x)** и вершины **v** не изменяются. Конец шага 7).
         II. **r(v)≤r(v`)**. Ранг Δ-дуги **(v,x)** увеличивается: **r`(v,x)=r(v`)+1**. Описатель Δ-дуги изымаем из списка Δ-дуг ранга **r(v,x)** и помещаем в список Δ-дуг

ранга **r`(v,x)**, при необходимости создавая описатель этого ранга. Если список Δ-дуг ранга **r(v,x)** стал пуст, уничтожаем описатель этого ранга и меняем ссылку из описателя вершины **v** на следующий ранг в списке рангов. Анализируем ранг вершины **v**.

i. Если список рангов Δ-дуг пуст, то есть, **r`(v)** неопределен, или **r`(v)≥N**, алгоритм останавливается (*стоп 2*).
ii. В противном случае, конец шага 8).

Лемма 5.1: Ранг пройденной Δ-дуги без петель меньше или равен увеличенному на **1** максимуму из определенных рангов концов ее пройденных дуг.

Когда ранг Δ-дуги устанавливается впервые (п.1.B.a и п.1.B.b.II) или корректируется впоследствии (2.B.b.II) он становится равным увеличенному на **1** максимуму из определенных рангов концов ее пройденных дуг. Поскольку ранги Δ-дуг и, тем самым, вершин не уменьшаются, между коррекциями ранга Δ-дуги ранги концов ее дуг не уменьшаются. Лемма доказана. □

Лемма 5.2: Если в компоненте сильно-Δ-связности **K** имеется непомеченная вершина, то для любой вершины **a∈VK** с рангом **r(a)>0** имеется вершина **b∈VK** с рангом **r(b)=r(a)–1**.

Допустим противное: для некоторой вершины **a∈VK** ранга **r(a)>0** любая вершина **b∈VK** имеет ранг **r(b)≠r(a)–1**. Рассмотрим множество **H** вершин, имеющих ранг **r(a)** или больше. Поскольку в **K** есть непомеченные вершины ранга **0**, **H** – непустое собственное подмножество **VK**. Поскольку **K** – компонент сильно-Δ-связности, **H** не может быть Δ-изолированным, то есть, существует Δ-дуга **(v,x)** с началом **v∈H**, концы всех дуг которой принадлежат **VK\H**. Эта Δ-дуга, очевидно, пройдена (ее начало **v** имеет ранг больше **0**), и концы всех ее пройденных дуг имеют ранги меньше **r(a)-1**. Но тогда, по Лемме 5.1, **r(v,x)≤ r(a)-1** и **r(v)≤r(v,x)≤r(a)-1**, что противоречит **v∈H**. Лемма доказана. □

Лемма 5.3: Когда алгоритм останавливается в вершине **v**, компонент сильно-Δ-связности **K(v)** обойден по стимулам (все Δ-дуги, выходящие из его вершин, пройдены).

Если произошел *стоп1*, то все пройденные вершины помечены.

Если произошел *стоп 2*, то помечены все пройденные вершины компонента **K(v)**. Действительно, это происходит в двух случаях. 1) все Δ-дуги, выходящие из **v**, имеют петли; 2) ранг вершины **v** слишком большой **r(v)≥N**. В случае 1) **v** – единственная вершина в **K(v)** и она полностью пройдена. В случае 2), очевидно, **r(v)≥N(v)**, где **N(v)** - число пройденных вершин компонента **K(v)**. Если бы в **K(v)** оставались непомеченные вершины, то, по Лемме 5.2, в **K(v)** были бы вершины с рангами **r(v), r(v)–1,…,0**, но тогда получилось бы, что **N(v)≥r(v)+1≥N(v)+1**, чего быть не может.

Таким образом, в обоих случаях остановки алгоритма все пройденные вершины компонента **K(v)** помечены, то есть, пройдены все выходящие из них Δ-дуги. Если в **K(v)** остались непройденные Δ-дуги, то они выходили бы из непройденных вершин **K(v)**, но тогда множество пройденных вершин **K(v)**, было бы непустым собственным подмножеством множества **VK(v)** и Δ-изолированным, чего быть не может. Следовательно, **K(v)** обойден по стимулам. □

Гарантированный Δ-обход графа 2-го рода с начальной вершиной из первого компонента. В таком графе алгоритм проходит путь из начальной вершины в последний компонент, и, если остановится, то, по Лемме 5.3, обойдет его по стимулам. Доказательство остановки алгоритма на любом графе дано ниже.

Лемма 5.4: Ранг вершины меньше **n**, а ранг Δ-дуги не превосходят **n**.

Ранг вершины до остановки алгоритма меньше **N** (*стоп 2*). Поскольку **N≤n**, ранг вершины меньше **n**. Учитывая Лемму 5.1, ранг Δ-дуги не превосходит **n**. □

Остановка алгоритма на любом графе с длиной пройденного маршрута **O(nm)**. Определим ранг Δ-дуги с петлей равным **n** и рассмотрим сумму $\mathbf{S = R_\Delta - r(v) + nL}$, где $\mathbf{R_\Delta}$ – сумма рангов пройденных Δ-дуг (включая Δ-дуги с петлями), **r(v)** – ранг текущей вершины. Поскольку **L≤N≤n≤m–1** и учитывая Лемму 5.4, **S =O(nm)**. На каждом шаге алгоритм проходит не более чем одну дугу. Поэтому для доказательства утверждения достаточно показать, что на каждом шаге (кроме, быть может, последнего) сумма **S** увеличивается хотя бы на **1**: **S`≥S+1**. Покажем это для всех 8 видов конца шага.

Конец шага 1). $\mathbf{R`_\Delta=R_\Delta}$, **v`=v**, **r`(v)<n**, **r(v)=0**, **L`=L+1**. Поэтому **S`–S = (–r`(v)+r(v)) + n ≥ 1**.
Конец шага 2). $\mathbf{R`_\Delta=R_\Delta+1}$, **r`(v`)=r(v)=0**, **L`=L**. Поэтому **S`–S = 1**.
Конец шага 3). $\mathbf{R`_\Delta=R_\Delta+r`(v,x)}$, **r`(v,x)=n**, **v`=v**, **r`(v)=r(v)**, **L`=L**. Поэтому **S`–S = n ≥ 1**.
Конец шага 4). $\mathbf{R`_\Delta=R_\Delta+r`(v,x)}$, **r`(v,x)=r(v`)+1**, **r`(v`)=r(v`)**, **r(v)=0**, **L`=L**. Поэтому **S`–S = r`(v,x)–r`(v`) = 1**.
Конец шага 5). $\mathbf{R`_\Delta=R_\Delta}$, **r`(v`)=0**, **r(v)>0**, **L`=L**. Поэтому **S`–S = r(v)–r`(v`) ≥ 1**.
Конец шага 6). $\mathbf{R`_\Delta=R_\Delta+r`(v,x)–r(v,x)}$, **r`(v,x)=n**, **r(v,x)=r(v)<n**, **v`=v**, **r`(v`)<n**, **r(v)>0**, **L`=L**. Поэтому **S`–S = r`(v,x)–r(v,x)+r(v)–r`(v`) = n–r(v)+r(v)-r`(v`) ≥ 1**.
Конец шага 7). $\mathbf{R`_\Delta=R_\Delta}$, **r`(v`)=r(v`)**, **r(v)>r(v`)**, **L`=L**. Поэтому **S`–S = r(v)–r`(v`) ≥ 1**.
Конец шага 8). $\mathbf{R`_\Delta=R_\Delta+r`(v,x)–r(v,x)}$, **r`(v,x)=r(v`)+1**, **r(v,x)=r(v)<n**, **r`(v`)=r(v`)**, **r(v) ≤r(v`)**, **L`=L**. Поэтому **S`–S = r`(v,x)–r(v,x)+r(v)–r`(v`) = r(v`)+1–r(v)+r(v)–r(v`) = 1**.

Время работы алгоритма. На каждом шаге неконстантное число элементарных операций тратится только на 1) поиск идентификатора вершины (конца пройденной дуги) среди идентификаторов пройденных вершин и 2) перемещение Δ-дуги из одного списка в другой при увеличении ее ранга.

Для 2) достаточно заметить, что, поскольку ранг Δ-дуги не уменьшается и не превосходит **n**, суммарно по всем шагам она перемещается не более чем на **n** позиций. Поэтому на перемещение всех Δ-дуг суммарно по всем шагам требуется не более **O(nm)** элементарных операций.

Время 1) зависит от операций сравнения, которые определены для идентификаторов вершин. Если определена только операция сравнения на равенство, мы тратим **O(n)** сравнений на каждый поиск, число которых **O(nm)**, то есть, всего $\mathbf{O(n^2m)}$ сравнений. Если определена также операция сравнения на больше/меньше, то вместо простого списка описателей вершин мы можем организовать сбалансированное дерево таких описателей. Поиск в таком дереве требует $\mathbf{O(\log_2 n)}$ сравнений, таких поисков **O(nm)** и всего имеем $\mathbf{O(nm\log_2 n)}$ сравнений. Коррекция дерева при добавлении в него новой вершины требует $\mathbf{O(\log_2 n)}$ операций, число таких коррекций **O(n)** и всего имеем $\mathbf{O(n\log_2 n)}$ операций.

Требуемая память. Поскольку ранг не превосходит **n**, значение ранга и ссылка по списку рангов занимают $\mathbf{O(\log_2 n)}$ бит. Ссылка по списку Δ-дуг занимает $\mathbf{O(\log_2 m)}$ бит. Пусть **I** и

X – размер в битах, соответственно, идентификатора вершины и стимула. Тогда описатель вершины занимает $\mathbf{O(I+\log_2 n)}$ бит, описатель ранга – $\mathbf{O(\log_2 n+\log_2 m)}$ бит, описатель Δ-дуги – $\mathbf{O(X+\log_2 m)}$ бит. Поскольку $\mathbf{n \le m-1}$, а с каждым описателем ранга связан непустой список Δ-дуг, суммарно получаем $\mathbf{O(nI+mX+m\log_2 m)}$ бит. Если поддерживается сбалансированное дерево описателей вершин, то на каждую ссылку по дереву требуется $\mathbf{O(\log_2 n)}$ бит памяти, а всего $\mathbf{O(n \log_2 n)}$ бит, что не изменяет порядок общей оценки.

На этом доказательство Теоремы 5.2 закончено. □

5.3. Модификации алгоритма

Для свободного алгоритма $\mathbb{B}_1$ легко построить его неизбыточную версию $\mathbb{B}_2$: вершина помечается при проходе по последней выходящей из нее непройденной Δ-дуге. Теперь посмотрим, какие достоверные вердикты могут выносить свободный алгоритм $\mathbb{B}_1$ и неизбыточный алгоритм $\mathbb{B}_2$. Если в момент остановки алгоритма отсутствуют непомеченные вершины, множество пройденных вершин Δ-изолировано и алгоритм совершил обход по стимулам пройденного графа. Для $\mathbb{B}_1$ достоверным будет вердикт «*если граф сильно-Δ-связен, то совершен гарантированный обход по стимулам*» (напомним, что мы рассматриваем только Δ-достижимые графы), а для $\mathbb{B}_2$ – более сильный вердикт «*если граф 2-го рода, то совершен гарантированный обход по стимулам*». Если же в момент остановки остались непомеченные вершины, то свободный алгоритм $\mathbb{B}_1$ не знает, являются они полностью пройденными или нет. Можно лишь утверждать, что совершен обход по стимулам компонента сильно-Δ-связности пройденного графа, в котором алгоритм остановился, и множество вершин этого компонента Δ-изолировано. Неизбыточный алгоритм $\mathbb{B}_2$, напротив, точно знает, что обход по стимулам не совершен.

Для более точных вердиктов необходимо анализировать пройденный граф. В изложенной выше версии алгоритма пройденный граф, фактически не сохраняется: мы запоминаем только пройденные вершины и стимулы пройденных Δ-дуг. Легко модифицировать алгоритм так, чтобы он сохранял информацию о всех пройденных дугах. Это не повлечет изменения порядка длины маршрута и времени работы алгоритма, но, естественно, увеличит требуемую память.

Рассматриваемые алгоритмы могут применяться также для Δ-покрытия любых Δ-достижимых графов с помощью повторных запусков алгоритмов с сохранением информации от предыдущей работы. Иначе говоря, алгоритмы могут работать с графами, для которых определена достоверная операция **reset** [9], которую они будут использовать только в случае необходимости – вместо остановки *стоп 2* (в частности, не будут использовать на сильно-Δ-связных графах).

Алгоритм $\mathbb{B}_2$ можно модифицировать (обозначим его $\mathbb{B}_3$) так, что он сможет гарантированно обходить по стимулам также графы 1-го рода, если ему каким-то внешним образом поставляется информация о связующих Δ-дугах. Пусть в каждой вершине графа задан предикат от стимула $\boldsymbol{\pi}\mathbf{(x)}$, который можно назвать *предикатом связующих Δ-дуг*. Предикат *достоверен*, если он истинен на и только на стимулах связующих Δ-дуг. Алгоритм, с внешними операциями **next**, **call** и $\boldsymbol{\pi}$, конечно, не является неизбыточным, но в некотором смысле «минимально избыточным» алгоритмом. Алгоритм $\mathbb{B}_3$ отличается от $\mathbb{B}_2$ тем, что сначала проходятся только несвязующие Δ-дуги, выходящие из пройденных вершин, а стимулы связующих Δ-дуг запоминаются в их начальных вершинах. Когда несвязующих непройденных Δ-дуг больше не остается и все

вершины помечены, алгоритм считает непомеченными начала связующих Δ-дуг и соответствующим образом корректирует ранги вершин и Δ-дуг (для чего просматривается пройденный граф).

Предикат π формально определен на тройках (граф, вершина, стимул). Будем называть предикат *неизбыточным*, если он не зависит от графа. Более точно, зависимость предиката от графа сводится к зависимости от множества стимулов, допустимых в вершине, то есть, формально предикат определен на тройках (вершина, множество допустимых в вершине стимулов, стимул). Если неизбыточный предикат рассматривать не как внешнюю, а как внутренную операцию алгоритма, то соответствующим образом модифицированный алгоритм $\mathbb{B}_3$ (обозначим его $\mathbb{B}_4$), во-первых, будет неизбыточным, и во-вторых, гарантированно обходить по стимулам все графы 2-го рода и те графы 1-го рода, на которых предикат достоверен. Вместе с тем, неизбыточный предикат, очевидно, не может быть достоверен на всех графах с выделенными начальными вершинами $\mathbf{v_0}$, изоморфных с точностью до раскраски Δ-дуг стимулами, если это не графы 2-го рода.

Еще одной областью применения алгоритма с предикатом (избыточным или неизбыточным) связующих Δ-дуг являются многоуровневые графы. Двухуровневый граф можно определить как граф 2-го уровня, в котором вершины являются графами 1-го уровня, причем все эти графы сильно-Δ-связны. Более строго, двухуровневый граф можно определить как граф, в котором некоторые Δ-дуги промаркированы. Если маркированные Δ-дуги удалить, то получится набор сильно-Δ-связных графов – графов 1-го уровня. Если факторизовать двухуровневый граф по отношению взаимной Δ-достижимости вершин через *немаркированные* Δ-дуги, то получится сильно-Δ-связный граф 2-го уровня. Если предикат понимать как предикат маркированных Δ-дуг, то алгоритм будет обходить двухуровневый граф по уровням: попадая первый раз в некоторый граф 1-го уровня, он сначала полностью его обходит по стимулам немаркированных Δ-дуг, а потом по маркированной Δ-дуге переходит в следующий граф 1-го уровня. При повторном вхождении в граф 1-го уровня, в нем достаточно пройти путь до нужной маркированной Δ-дуги, выходящей из графа. **p**-уровневый граф определяется по индукции как двухуровневый граф, компонентами которого являются **p–1**-уровневые графы. Алгоритм можно модифицировать для работы с любым наперед заданным числом уровней **p**. Длина алгоритмического обхода **p**-уровневого графа во многих случаях приближается к оптимальной, которая меньше, чем **O(nm)**.

6. Заключение

Предложенные в настоящей статье свободные и неизбыточные алгоритмы обхода по стимулам графов и тестирование на их основе были разработаны и апробированы, начиная с 1995 г., группой RedVerst [1] в ходе выполнения нескольких масштабных проектов по тестированию разнообразного программного обеспечения [17,12], которое велось на основе функциональных спецификаций, полученных на стадии проектирования или реинжиниринга.

Как правило, при тестировании критичным является не столько сложность по времени и памяти алгоритма Δ-обхода, сколько число тестовых воздействий, то есть, длина Δ-обхода. Свободный алгоритм $\mathbb{B}_1$ (и его неизбыточная версия $\mathbb{B}_2$), имеющие хорошие оценки сложности, обеспечивают лишь минимальный порядок **nm** длины Δ-обхода в наихудшем случае. В то же время на многих графах с минимальной длиной Δ-обхода по порядку меньшей **nm**, они могут строить Δ-обход столь же длинный, как и в наихудшем случае. Изучение неизбыточных алгоритмов, стремящихся совершить Δ-обход

минимальной длины, представляется полезной задачей для будущих исследований. Здесь можно указать на аналогичные результаты для детерминированных графов (см. например, [14]).

Литература

1. http://www.ispras.ru/~RedVerst/
2. R. Milner. A Calculus of Communicating Systems, LNCS, vol. 92, Springer-Verlag, 1980.
3. J. R. Burch, E. M. Clark, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In Design Automation Conference, pp. 46-51, 1990
4. S. Fujiwara and G. von Bochmann. Testing Nondeterministic Finite State Machine with Fault Coverage. IFIP Transactions, Proceedings of IFIP TC6 Fourth International Workshop on Protocol Test Systems, 1991, Ed. by Jan Kroon, Rudolf J. Hei-jink, and Ed Brinksma, 1992, North-Holland, pp. 267-280.
5. G. Luo, A. Petrenko, G. von Bochmann. Test Selection based on Communicating Nondeterministic Finite State Machines using a Generalized Wp-Method, IEEE Transactions, vol. SE-20, No. 2, 1994.
6. G. von Bochmann, A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. Proceeding of ISSTA 1994, pp. 109-124.
7. G. M. Swamy, V. Singhal, and R. K. Brayton. Incremental methods for FSM Traversal. Technical Report, Electronics Research Lab, University of California, Berkeley, CA94720, 1995.
8. G. Cabodi, L. Lavagno, E. Macii, M. Poncino, S. Quer, P. Camurati, E. Sentovicha. Enhancing FSM Traversal by Temporary Re-Encoding. Proceedings of IEEE International Conference on Computer Design, pp. 6-11, 1996.
9. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. Proceedings of the IEEE, vol. 84, 8:1090-1123, Berlin, IEEE Computer Society Press, Aug. 1996.
10. A. Petrenko, N. Yevtushenko, G. von Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. Selected proceedings of the IFIP TC6 9-th international workshop on Testing of communicating systems, September 1996.
11. Marine Tabourier, Ana Cavalli and Melania Ionescu , A GSM-MAP Protocol Experiment Using Passive Testing, Proceeding of FM'99 (World Congress on Formal methods in development of Computing Systems), Toulouse (France), 20-24 September 1999.
12. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM’99: Formal Methods. LNCS, vol. 1708, pp. 608621, Springer-Verlag, 1999.
13. И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин. Использование конечных автоматов для тестирования программ. "Программирование", 2000, №2.
14. S. Albers and M.R. Henzinger, Exploring Unknown Environments, SIAM J. Comput., Vol. 29, No. 4, 2000, pp. 1164-1188.
15. Yuri Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111.
16. M. Barnett, L. Nachmanson, and W. Schulte. Conformance Checking of Components Against Their Non-deterministic Specifications. Microsoft Research Technical Report MSR-TR-2001-56.
17. I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002.
18. И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированнный случай. «Программирование», 2003, №???, стр.???