Санкт-Петербургский государственный университет информационных технологий, механики и оптики

На правах рукописи

Казаков Матвей Алексеевич

Методы построения визуализаторов алгоритмов дискретной математики на основе автоматного подхода

Специальность 05.13.06 – «Автоматизация и управление технологическими процессами и производствами (образование) »

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель –
доктор технических наук,
профессор В.Г. Парфенов

Санкт-Петербург
2010

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность проблемы. С вводом Федеральных государственных образовательных стандартов (ФГОС) [1] усиливается роль самостоятельной работы студентов (СРС) [1], что требует разработки новых образовательных технологий для ее поддержки. Кроме того ФГОС содержат требование к обеспечению не менее 10–20% аудиторных занятий студентов в интерактивной форме. Это обстоятельство ставит новые задачи по разработке интерактивных средств обучения, призванных индивидуализировать образовательный процесс массовой подготовки выпускников вузов. При уровневой подготовке выпускников вузов информационной направленности важное место занимают методы и алгоритмы дискретной математики, изучение которых создает базис (базовые знания, умения и навыки) для формирования различных универсальных (общекультурных) и профессиональных навыков выпускников. Уже сейчас во многих вузах РФ применяются виртуальные лаборатории, тренажеры и электронные тьюторы для поддержки практических и самостоятельных занятий по изучению алгоритмов дискретной математики. Многие годы эти алгоритмы изучались по книгам, в которых практически невозможно было отразить динамику [2 – 6]. В восьмидесятых годах двадцатого века в нескольких американских университетах [7], а в конце девяностых годов также и в Санкт-Петербургском государственном университете информационных технологий, механики и оптики (СПбГУ ИТМО) [8, 9, 78, 80] и Санкт-Петербургском государственном университете (СПбГУ) [10, 11] пришли к выводу, что алгоритмы следует изучать не только в статике в виде диаграмм, но и в динамике (в процессе работы). Для этого необходимо строить специальные программы, которые должны представлять алгоритмы в удобной для обучающихся форме: наряду с текстовыми комментариями должны отображаться динамические иллюстрации. Программы для реализации динамических иллюстраций, были названы *визуализаторами*. В течение последних 10 лет проблемами построения визуализаторов занимались

многие специалисты в СПбГУ ИТМО, [12] и СПбГУ [10, 11, 13], в том числе автор настоящей диссертации [76, 78]. Однако все эти работы обладали существенным недостатком: визуализаторы для каждого алгоритма индивидуально и отсутствовали какие-либо методы их формализованной реализации.

С начала девяностых годов в России (в частности, в СПбГУ ИТМО) разрабатывался метод автоматного программирования (программирование с явным выделением состояний [44]), суть которого состоит в применении конечных автоматов в программировании. В 2001 г. автором настоящей работы было высказано предположение, что формализованные методы построения визуализаторов могут базироваться на автоматном подходе. Это предположение оказалось верным, что и будет показано в настоящей диссертации.

В диссертации предлагаются методы формализованного построения визуализаторов алгоритмов дискретной математики на основе автоматного подхода. Эти методы были внедрены автором в учебный процесс, который осуществлялся и осуществляется на кафедре «Компьютерные технологии» СПбГУ ИТМО. На основе этого подхода были созданы десятки визуализаторов алгоритмов программирования. В дальнейшем эти методы были автоматизированы, что потребовало разработки дополнительных технологий, которые учитывают особенности процесса автоматизации реализации визуализаторов. Эта работа выполнялась совместно с Г.А. Корнеевым [74], который свою диссертационную работу посвятил особенностям автоматизации построения визуализаторов. Настоящая работа описывает методы, которые, во-первых, весьма эффективны при их использовании вручную, а во-вторых, легли в основу автоматизации построения визуализаторов, описанной в диссертации Г.А. Корнеева. Опыт преподавания в Интернет-школе программирования, а также студентам первых и вторых курсов СПбГУ ИТМО показывает, что для хорошего понимания алгоритмов дискретной математики необходимо как ручное, так и автоматизированное построение визуализаторов. Ручное

построение позволяет глубже как понять природу самих алгоритмов, так и природу процессов визуализации алгоритмов.

Кроме того, ручные методы являются эффективным средством обучения интерактивному программированию, также являются простейшей технологией программирования, которую сравнительно легко осваивают студенты младших курсов. Использование автоматизации позволяет, во-первых, быстро создавать визуализаторы для сложных алгоритмов (за счет реализации алгоритмов посредством системы взаимодействующих автоматов), во-вторых, расширить функциональные возможности визуализаторов алгоритмов по сравнению с построением визуализаторов вручную (откат назад в итеративных алгоритмах и визуализация рекурсивных алгоритмов), а, в-третьих, проверять корректность построения визуализаторов. Кроме перечисленных достоинств автоматизированный подход обладает существенным недостатком: автоматическое выделение состояний приводит к очень большому их числу, что делает таким образом автоматы ненаглядными. Это исключает возможность использования указанного метода для ручного построения визуализаторов [14].

Конечные автоматы известны с пятидесятых годов прошлого века. Они в основном применялись для компиляторов и протоколов. Наиболее близкой к теме диссертации является применение автоматов для реализации пользовательского интерфейса. В 2007 г. ведущий сотрудник компании *IBM* Э. Принг предложил использовать автоматы для реализации виджетов [15 – 17], в частности, для реализации всплывающих окон. Однако в силу того, что методы формализованного перехода в его работах отсутствовали, были допущены ошибки при построении конечных программ, что отмечено в работе [18]. В этой же работе было отмечено, что методы, описываемые в настоящей диссертации, были ранее рассмотрены автором диссертации в работе [69].

Из изложенного следует, что для целей обучения программированию ручное построение визуализаторов более эффективно. Поэтому разработка

методов ручного формализованного построения визуализаторов алгоритмов является актуальной.

Цель диссертационной работы – разработка методов построения визуализаторов алгоритмов дискретной математики на основе автоматного подхода. Для достижения данной цели были поставлены и решены **следующие задачи**.

1. Анализ возможностей автоматного подхода для решения проблемы построения визуализаторов алгоритмов дискретной математики.
2. Разработка формализованного метода построения визуализаторов алгоритмов на основе конечных автоматов.
3. Разработка метода построения визуализаторов алгоритмов на основе системы взаимодействующих автоматов.
4. Разработка метода простой анимации алгоритмов при построении визуализаторов.
5. Исследование разработанных методов и внедрение результатов работы в учебный процесс.

Научная новизна. На защиту выносятся результаты, обладающие научной новизной.

1. Подход к реализации визуализаторов алгоритмов дискретной математики на основе конечных автоматов.
2. Метод построения визуализаторов на основе конечных автоматов.
3. Метод построения визуализаторов на основе системы взаимодействующих автоматов.
4. Метод для обеспечения простой анимации визуализаторов на основе автоматного подхода.

Методы исследования. В работе использованы методы теории автоматов, дискретной математики и теории алгоритмов.

Достоверность научных положений, выводов и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием постановок задач, точной формулировкой критериев,

компьютерным моделированием, а также результатами внедрения методов, предложенных в диссертации, на практике.

Практическое значение. Результаты, полученные в диссертации, используются на практике:

1. В СПбГУ ИТМО на кафедре «Компьютерные технологии» при разработке визуализаторов для образовательного портала «Интернет-школа программирования».
2. В учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсу «Теория автоматов в программировании».
3. В СПбГУ ИТМО на кафедре «Компьютерные технологии» при разработке визуализаторов в рамках учебного курса «Дискретная математика» [12].
4. Данные методы являются основой для построения многих других визуализаторов алгоритмов дискретной математики.

Апробация результатов работы. Основные положения диссертационной работы докладывались на научно-методических конференциях: «Телематика-1999» (СПб.), «Дистанционное обучение. Проблемы и перспективы взаимодействия вузов Санкт-Петербурга с регионами России» (СПб., 1999), «Телематика-2000», «Телематика-2001», «Телематика-2002», «Телематика-2003» (СПб.), «Конференция молодых ученых СПбГУ ИТМО» (СПб., 2004), «Телематика-2004» (СПб.), «Телематика-2005» (СПб.), «Профессиональное образование, наука, инновации в XXI веке» (СПб., 2007), «Научное программное обеспечение в образовании и научных исследованиях» (СПб., 2008).

Внедрение результатов работы. Результаты диссертации использованы при выполнении следующих научно-исследовательских работ: «Разработка технологии программного обеспечения систем управления на основе автоматного подхода» – гос. контракт № 10038 по заказу Министерства образования РФ в 2001 – 2005 гг. (http://is.ifmo.ru/science/1/), и «Разработка

технологии автоматного программирования», выполненной по гранту Российского фонда фундаментальных исследований № 02-07-90114 в 2002 – 2003 гг. (http://is.ifmo.ru/science/2/).

Публикации. По теме диссертации опубликовано 28 научных работ, из них 21 печатная работа, в том числе 11 статей, из которых пять статей опубликованы в журналах из перечня ВАК, 7 отчетов по научно-исследовательской работе.

Награды. В 2008 году автор стал лауреатом премии Правительства Российской Федерации в области образования в составе авторского коллектива научно-практической разработки «Инновационная система поиска и подготовки высококвалифицированных специалистов в области производства программного обеспечения на основе проектного и соревновательного подходов». В рамках этого проекта автор работал над разработкой методов построения визуализаторов и созданием Интернет-школы программирования. В 2008 году эта разработка стала лауреатом премии Правительства Российской Федерации в области образования (http://www.rg.ru/2009/01/16/premii-obrazovanie-dok.html).

Структура диссертации. Диссертация изложена на 178 страницах и состоит из введения, пяти глав и заключения. Список литературы содержит 97 наименований. Работа иллюстрирована 96 рисунками и 7 таблицами.

Глава 1 содержит описание проблемы построения визуализаторов, обосновывает несостоятельность существующих подходов. В главе приведены результаты исследований, предшествующих диссертации, описываются проблемы построения визуализаторов и формируются задачи для решения в рамках диссертационной работы.

В главе 2 обосновывается целесообразность использования автоматного подхода при построении визуализаторов. Далее в этой главе предлагается формализованный метод построения визуализаторов алгоритмов, рассмотрены две модификации метода – на основе автоматов Мили и Мура. Предлагаемый метод иллюстрируется двумя разобранными примерами. В заключение главы

приводится сравнение традиционного, эвристического автоматного и формализованного автоматного подходов.

В главе 3 предлагается метод построения визуализаторов алгоритмов на основе системы конечных автоматов. Предложенный метод расширяет возможности по созданию сложных визуализаторов за счет разбиения визуализаторов на уровни детализации.

В главе 4 предлагается метод простой анимации для визуализаторов алгоритмов. Этот метод позволяет реализовывать плавное отображение хода алгоритма. Метод снабжен двумя примерами, что подтверждает его универсальность.

В главе 5 приводится обзор практического внедрения результатов диссертации в учебный процесс, приводятся результаты статистических исследований трудозатрат на построение визуализаторов.

ГЛАВА 1. ВИЗУАЛИЗАТОРЫ АЛГОРИТМОВ ДИСКРЕТНОЙ МАТЕМАТИКИ

В настоящее время широкую популярность приобрел такой образовательный инструмент, как визуализаторы алгоритмов [9]. Визуализаторы алгоритмов в некоторых университетах являются неотъемлемой частью процесса обучения дискретной математике, а также дистанционного обучения теории алгоритмов [77].

В первой главе проведем обзор предметной области и сформулируем задачи, которые будут решаться в рамках данной работы.

В разд. 1.1 приведен общий обзор визуализаторов алгоритмов, описаны общие свойства визуализаторов. При этом в разд. 1.2 описывается роль визуализаторов в образовательном процессе и, в частности, в дистанционном обучении. В разд. 1.3 проводится обзор существующих систем визуализации и подходов к построению визуализаторов. Разд. 1.4 акцентирует внимание на предварительных исследованиях автора в области традиционного эвристического построения визуализаторов. Разд. 1.5 посвящен разъяснению технологической составляющей построения визуализаторов. В разд. 1.6 приведен обзор автоматного программирования и указаны достоинства автоматного программирования при построении визуализаторов. Разд. 1.7 завершает первую главу формулировкой задач решаемых в рамках данной диссертационной работы.

1.1. Визуализаторы алгоритмов

Дискретная математика – наука, изучающая область математики, занимающаяся изучением дискретных структур, которые возникают как в пределах самой математики, так и в её приложениях. [19]. Одним из ключевых разделов дискретной математики является *теория алгоритмов*. Эта теория рассматривает общие свойства и закономерности алгоритмов и разнообразные формальные модели их представления [20].

Одной из актуальных проблем в обучении дискретной математике и программированию является процесс изучения вычислительных алгоритмов. Автор настоящей диссертации занимается проблемами обучения дискретной математике с 1997 г. [75 – 78].

Проблема обучения алгоритмам дискретной математики существует с начала их использования в вычислительной технике. Сначала алгоритмы изображались в текстовом виде, потом – в виде схем алгоритма (блок-схем), а затем в виде псевдокода. Современные книги описывают алгоритмы в виде набора примеров входных и выходных данных, а также исходного кода алгоритма, реализованного на одном из императивных языков программирования, таких как *C*/*C*++, *Pascal* или *Java* [21]. *Императивным программированием* называется такое программирование, в котором программа состоит из набора инструкций, выполняемых в определенной последовательности [22].

Процесс образования при использовании для изучения алгоритмов книг происходит одним из следующих способов:

1. Статическое восприятие текста с динамической прокруткой в голове.
2. Реализация алгоритма на выбранном языке программирования, либо копирование алгоритма из книги с тем, чтобы динамически, шаг за шагом, отследить действие алгоритма на тестовых примерах.

Первый способ является достаточно сложным для большинства учащихся, поскольку требует программистского воображения. Он доступен лишь опытным программистам, а не учащимся старших классов школы или младших курсов института, только приступающим к изучению алгоритмов дискретной математики. Второй способ является более прямолинейным и доступным, однако акцентирует внимание не на сути алгоритма, а на его программной реализации и тонкостях используемого языка программирования. Таким образом, традиционное статическое изложение материала по дискретной математике является неэффективным с точки зрения изучения алгоритмов. Кроме того, если рассмотреть приведенные способы изучения с точки зрения

использования на лекциях по дискретной математике, то очевидно, что они также не эффективны. Если первый способ требует определенных усилий, то второй и вовсе невозможно реализовывать на лекционных занятиях.

Дальнейшие исследования в области преподавания дискретной математики привели к возникновению в середине восьмидесятых годов [7, 25] нового подхода к преподаванию – появились визуализаторы алгоритмов дискретной математики.

Визуализатор – это программа, иллюстрирующая выполнение алгоритма при определенных входных данных. Примерами визуализаторов могут служить, например, программа, занимающаяся построением графика функции, либо программа, визуально моделирующая какой-либо физический процесс. Применительно к дискретной математике и программированию, визуализаторы обычно моделируют некоторые алгоритмы, давая возможность обучающемуся при помощи интуитивно понятного интерфейса проходить алгоритм шаг за шагом от начала до конца, а при необходимости, и обратно [43].

Перечислим отличительные характеристики визуализаторов.

1. Простота использования, определяемая понятностью интерфейса. Поэтому для работы с визуализатором обычно не требуется специальная подготовка.
2. Четкость и простота представления визуализируемого процесса.
3. Компактность визуализаторов. Это при необходимости упрощает передачу визуализаторов в сети Интернет, что особенно важно при дистанционном обучении.

Визуализаторы в рассматриваемой области решают следующие задачи, возникающие в процессе обучения.

1. Графическое и текстовое разъяснение действий алгоритма на конкретных наборах входных данных. При этом понимание алгоритма не требуется, поскольку именно визуализатор должен объяснить действие алгоритма.

2. Предоставление пользователю инструмента, реализующего данный алгоритм. В результате учащийся освобождается от необходимости выполнять шаги алгоритма, так как их автоматически выполняет визуализатор.

Визуализатор выполняет обычно следующие функции.

1. Пошаговое выполнение алгоритма.
2. Просмотр действия алгоритма при разных наборах входных данных, в том числе и введенных пользователем.
3. Просмотр действия алгоритма в динамике.
4. Перезапуск алгоритма на текущем наборе входных данных.

Динамическая визуализация наглядно демонстрирует такую характеристику алгоритма, как трудоемкость (особенно при пошаговой демонстрации). Для некоторых алгоритмов (например, машины Тьюринга) динамический вариант демонстрации вообще представляется более естественным, чем любой набор статических иллюстраций.

В задачи визуализатора алгоритма входит:

1. Отображение входных и выходных данных в наглядной форме – доходчивое представление абстрактного понятия данные в виде картинки. Такое представление необходимо, как для учащихся с начальным уровнем подготовки, так и для более продвинутых учащихся с целью упрощения восприятия.
2. Отображение внутренних служебных переменных.
3. Отображение процесса воздействия алгоритма на входные данные и внутренние переменные:
    a. Изменение данных.
    b. Копирование и перемещение данных.
    c. Процесс принятия решений.
4. Комментарии к каждому действию алгоритма.
5. Отображение работы алгоритма по шагам.

6. Возможность ввода данных с целью апробации алгоритма на разных наборах входных данных.

Начиная с 1997 г., автор настоящей диссертации совместно с другими преподавателями кафедры КТ занимается проблемой построения визуализаторов.

1.2. Визуализаторы в дистанционном обучении

Одной из важных предпосылок к началу работы над методами создания визуализаторов является опыт работы над проектом дистанционного обучения [23, 24] «Интернет-школа программирования» [26, 76, 77]. Этот проект разрабатывался совместно с сотрудниками и преподавателями кафедры «Компьютерные технологии» СПбГУ ИТМО.

Основной целью поставленной перед Интернет-школой программирования является преподавание дискретной математики и программирования. С целью разработки технологии преподавания были проведены исследования. Ниже приводятся выдержки из результатов этих исследований.

Основную часть лекционных курсов в рамках дисциплин дискретной математики и теории алгоритмов составляет обсуждение разнообразных алгоритмов обработки информации, представляемой различными структурами данных.

Систематический подход преподавателя к собственно процессу ведения занятий определяют несколько факторов:

- состав аудитории слушателей: школьники, студенты нематематической специализации, студенты-математики;
- техническая оснащенность лекционного помещения: доска + мел, доска + цветные фломастеры, наличие проекционной аппаратуры, возможность использования компьютера совместно с проекционной аппаратурой;

- затраты времени на подготовку иллюстративного лекционного материала: либо во время лекции – рисунки, графики и т. д. на доске или планшете, либо предварительно – плакаты, диапозитивы, компьютерные файлы.

Квалифицированный преподаватель, стремится решить одновременно две (частично конфликтующие) задачи: во-первых, обеспечить, по возможности, наилучший уровень подачи лекционного материала и, в то же время, минимизировать собственные трудозатраты по подготовке лекционного материала. Возможности решения обеих задач определяются вариантами совместимости указанных выше факторов.

Так, «беззатратный» вариант (без иллюстраций вообще) допустим (да и то не всегда) только в аудитории хорошо подготовленных студентов-математиков. В этом случае вполне можно свести обсуждение алгоритма к изложению «на пальцах», а затем уже формализовать его, описав входной и выходной наборы, шаг инициализации и стандартный шаг алгоритма. В любой иной аудитории более приемлем технологический подход, опирающийся на неоспоримый тезис «Лучше один раз увидеть, чем сто раз услышать». Приведем простой контрпример: лекция по дифференциальному исчислению, которая транслируется из радиостудии.

Итак, для обычной аудитории слушателей технология лекционного процесса определяется уже лишь двумя составляющими – факторами технической оснащенности и потенциальными трудозатратами на подготовку иллюстративного материала. В этом сочетании, полагаем, первичной следует считать техническую оснащенность лекционного помещения. Соответственно, имеем четыре варианта ведения лекционного занятия.

1. **Традиционная технология.** Иллюстрации к обсуждаемому алгоритму рисуются по ходу лекции на доске, а при несколько лучшей оснащенности – на планшете с проецированием их на демонстрационный экран. Неустранимые недостатки очевидны: во-первых, качество иллюстраций весьма зависит от художественных

способностей лектора и, во-вторых, время доступа обучаемых к иллюстративному материалу жестко ограничено продолжительностью занятия.

2. **«Улучшенная» традиционная технология.** В распоряжении лектора заранее подготовленный иллюстративный материал – плакаты, диапозитивы и т. п. Недостатки: «статичность» набора иллюстраций, ограничивающая набор примеров входного потока для обсуждаемого алгоритма; та же проблема со временем доступа к этому набору у слушателей.
3. **Технология «книга с картинками»**. Используются компьютер и демонстрационный экран; заранее подготовлен набор файлов-иллюстраций, например, в *JPG*- или *GIF*-формате; те же изображения могут быть встроены в *HTML*-файлы.

По-видимому, трудозатраты на подготовку иллюстраций сопоставимы с теми, что имеют место и в варианте два. При этом заметны достоинства этой технологии: для преподавателя – возможность «беззатратной» воспроизводимости лекции; для слушателей – доступность лекционных материалов не только во время занятия.

Естественно, такая технология вполне подходит для дистанционного обучения в рамках курсов дискретной математики. Мы апробировали такой подход, но довольно скоро поняли, что более рационален другой вариант. Он включает технологию предыдущего подхода, но дополнен возможностью использования динамических иллюстраций. Иначе говоря, иллюстративный материал, помимо текстовых файлов, включает визуализаторы алгоритмов. В качестве таковых выступают специальные программы, в процессе работы которых на экране динамически демонстрируется действие алгоритма на выбранном наборе данных. При этом доступны режимы использования входных наборов, заготовленных заранее, либо вводимых с клавиатуры, либо, наконец, генерируемых случайным образом.

В ряде тем (например, в теме «Сортировка массива») программа-визуализатор включает несколько родственных алгоритмов, что позволяет наглядно продемонстрировать как общий подход, так и различие в механизмах их действия. Особое место занимает проблема визуализации в дистанционном курсе обучения. Здесь целесообразно встраивать динамический визуализатор непосредственно в текст лекции (по аналогии с иллюстрацией в учебнике).

Поскольку визуализатор сопровождает текстовым комментарием каждый шаг алгоритма, то можно сказать, что он почти заменяет преподавателя. В дистанционном варианте заметно еще одно достоинство динамической визуализации: возможность регулировать ее скорость. Это значит, что обучаемый имеет возможность подобрать ее, руководствуясь собственными способностями усвоения материала. Наконец, при изучении работы некоторых алгоритмов оказывается полезным и режим «шаг назад».

С точки зрения возможностей подготовки и использования лекционных уроков по курсам дискретной математики и программирования, наш опыт показывает конкурентоспособность двух следующих технологий.

Если ставить целью снабдить, как слушателей, так и лектора, очным лекционным курсом, то в качестве языка для создания программ-визуализаторов хорошо подходит любая система программирования, включающая многообразные средства создания оболочек визуализаторов. Если же исходить из целей создания дистанционного курса, размещаемого на *Web*-сервере, то использование систем для ориентированных на создание обычных приложений становится невозможным. Визуализаторы, которые включаются в *HTML*-лекции, следует писать на языке *Java*.

При кафедре «Компьютерные технологии» СПбГУ ИТМО создана Интернет-школа программирования (*IPS*) [26]. Одним из направлений ее деятельности является дистанционное обучение школьников. В уроках-лекциях, размещенных на сервере http://ips.ifmo.ru и готовящихся к этому, визуализаторы алгоритмов используются весьма широко. Обычный «урок» включает статический текст, два–три визуализатора, поясняющих

рассматриваемые в уроке алгоритмы, а также набор задач, в том числе, предлагаемых для дистанционного тестирования [75, 76].

Исходя из опыта применения визуализаторов, можно с уверенностью утверждать, что их применение, как в очном, так и в дистанционном обучении весьма способствует улучшению понимания учащимися лекционного материала.

1.3. Подходы к построению визуализаторов

Как отмечено выше, одним из методических приемов [8, 26, 78, 80] при обучении программированию и дискретной математике является разработка визуализаторов алгоритмов [2, 3].

Анализ литературы [27, 28], а также реализации визуализаторов [29 – 32] показал, что технологии разработки визуализаторов рассматриваемого класса обычно являются эвристическими.

Опыт построения визуализаторов [8, 78] свидетельствует о том, что при эвристическом подходе каждый алгоритм требует индивидуальной реализации. Другими словами, формализованный метод разработки визуализаторов рассматриваемого типа отсутствовал, и основные успехи в этой области были педагогическими [8, 33, 78].

Детальное изучение российского и зарубежного опыта построения визуализаторов привело к выводу, что все подходы обладали существенным недостатком. Они создавались эвристически, и качество визуализатора зависело от личных предпочтений педагога [12]. Следует отметить, что кроме полностью эвристических визуализаторов алгоритмов, также существуют, так называемые, *системы визуализации*. Системы визуализации предоставляют библиотеки готовых элементов визуализаторов и элементы пользовательского интерфейса.

Первый зафиксированный визуализатор, демонстрирующий работу связанных списков, появился в далеком 1966 г. в *Bell Telephone Laboratories* [34, 35]. Лишь в конце восьмидесятых — начале девяностых были созданы первые системы визуализации: *BALSA* (*Brown ALgorithm Simulator and*

Animator) [25] и *TANGO* (*Transition-based Animation GeneratiOn*) [36]. В настоящее время существует большое число систем визуализации (например, *Animal* [37], *Leonardo* [38], *Zeus* [39], *CATAI* [40], *Mocha* [41]). Однако, поскольку системы визуализации по своей сути реализуют пользовательский интерфейс и различные визуальные эффекты, но не предоставляют автоматического или формализованного построения визуализаторов алгоритмов, то их использование не дает преимуществ по сравнению с эвристическим подходом. Следует отметить, что в настоящее время существует одна система визуализации, реализующая возможность формализованного построения визуализаторов по алгоритмам, разработанная Г.А. Корнеевым на кафедре «Компьютерные технологии» СПбГУ ИТМО [42, 43]. Данная система разрабатывалась Г.А. Корнеевым на основе идей автора изложенных в настоящей диссертации, а также совместной работы над разработкой методов формализованного перевода алгоритмов в визуализаторы [74]. Как уже отмечалось выше, работа Г.А. Корнеева была продолжена в области автоматизации. В то время как автор настоящей работы работал в области **ручного** построения визуализаторов, что является хоть и более трудоемким занятием, но гораздо более эффективным для обучения теории алгоритмов.

После анализа существующих алгоритмов и систем визуализации можно сделать следующий вывод. При ручном построении визуализаторов алгоритмов существует три подхода.

- **Традиционный эвристический**. При этом подходе визуализатор строится из общих соображений автора.
- **Автоматный эвристический**. Автоматный подход основан на описании поведения визуализатора алгоритма на основе автоматов.
- **Автоматный формализованный.** Визуализатор с использованием автоматов строится на основе алгоритма формализовано.

Автоматный формализованный метод, рассматриваемый в настоящей диссертации, превращает автоматный эвристический подход в метод формального построения визуализаторов.

Таким образом, задача разработки формализованного ручного метода для построения визуализаторов является актуальной.

1.4. Традиционный эвристический подход

В основе этого подхода лежит построение алгоритма на основе «общих соображений». При этом построение визуализатора происходит по следующей схеме:

1. Формулировка алгоритма в словесно-математической форме.
2. Принятие решения о визуализируемых элементах и процессе отображения визуализации.
3. Построение программы, реализующей визуализатор.

Данным способом строились визуализаторы с 1997 по 2001 гг. При этом в 1999 – 2001 гг. автором данной работы были организованы курсовые работы по построению визуализаторов для студентов первого курса. Опыт проведения этих работ показал, что из-за отсутствия метода разработки визуализаторов алгоритмов, создание каждого визуализатора требовало достаточно большого времени от преподавателя – от одного до двух рабочих дней (8 – 16 часов). Столь высокие затраты были вызваны большим числом корректировок и переделок начальных вариантов визуализаторов. В основном студенты, не имеющие опыта самостоятельной работы, не обладали достаточными знаниями и навыками для принятия правильных решений о формировании динамических иллюстраций. Кроме того, отсутствие формализованного подхода, требовало в каждом визуализаторе разработки уникального эвристического способа построения программы, что существенно замедляло процесс достижения финального результата. Ниже приведен список некоторых из этих курсовых работ (табл. 1).

Как следует из приведенных данных, эвристические подходы построения визуализаторов делятся условно на следующие классы.

- **Эвристика**. Суть подхода состоит в том, что визуализатор является неотъемлемой частью реализации алгоритма. Визуализация при этом

происходит за счет намеченных заранее остановок в процессе выполнения алгоритма и вывода всех переменных, с которыми оперирует алгоритм в виде иллюстрации. Далее переход к следующему шагу осуществляется в каждой ситуации особенным способом, зависящим от конкретного алгоритма. В результате такого подхода полученный визуализатор можно классифицировать как «неповторимое произведение искусства».

Таблица 1. Курсовые работы, выполненные под руководством автора

№	Студент	Алгоритм	Визуализатор
1.	Золотухин Ю.	*LZ*–сжатие	*Java*, стек состояний
2.	Добровольский В.	Перебор перестановок	*Java*, эвристика
3.	Крылов Р.	Обход дерева	*Java*, от предыдущей точки визуализации к следующей
4.	Михалев Е.	Генератор перестановок	*Java*, от предыдущей точки визуализации к следующей
5.	Прокушкин И.	Алгоритм Кнута-Морриса-Пратта	*Java*, стек состояний
6.	Филоненко Ф.	Сортировки массива	*Java*, стек состояний
7.	Порох Ю.	Поиск выпуклой оболочки	*Delphi*, от предыдущей точки визуализации к следующей
8.	Аничкин И.	Венгерский метод	*C++*, от предыдущей точки визуализации к следующей
9.	Альхов С.	Хэш-код	*Java*, эвристика
10.	Белов Д.	Сортировка фон Неймана	*Delphi*, эвристика
11.	Бородин Т.	Алгоритм Кнута-Морриса-Пратта	*Delphi*, стек состояний
12.	Наумов Л.	Алгоритмы поиска подстрок	*Delphi*, стек состояний

- **«От предыдущей точки визуализации к следующей»**. При таком подходе визуализатор разбивается на точки визуализации, которые отображаются при шагах алгоритма. При этом явно выделен метод перехода из предыдущей точки к следующей. Такой способ работает только в повторяющихся итеративных алгоритмах, где каждый шаг

повторяет предыдущий – логика алгоритма не зависит от текущего состояния переменных.

- **Стек состояний**. Этот способ является самым технологичным из эвристических, поскольку является универсальным. Суть способа состоит в том, что перед началом визуализации алгоритм запускается на входных данных и по ходу выполнения алгоритма в памяти запоминаются контрольные точки алгоритма. Эти точки содержат значения всех переменных, над которыми алгоритм производит действия. Метод является универсальным, поскольку может повторяться на любом алгоритме.

Анализ всех перечисленных выше эвристических методов построения визуализаторов выявляет следующие недостатки.

- Отсутствует формализованный подход к переходу от алгоритма к визуализатору.
- Отсутствует метод выделения контрольных точек, которые следует визуализировать.
- Отсутствует метод перехода из одной контрольной точки визуализатора к другой.

Отметим, что подход «Стек состояний» частично справляется с проблемой формализации, однако, по сути, он решает лишь проблему построения визуализатора как такового, но не решает проблему преобразования визуализатора в набор контрольных точек, что является самой большой проблемой из перечисленных выше. Именно выделение контрольных точек и является в эвристическом подходе элементом искусства и зависит в большой степени от квалификации автора визуализатора.

Из изложенного следует, что эвристический подход, хоть и имеет право на существование, что неоднократно подтверждено на практике, но методом не является и не гарантирует достижение результата при построении визуализаторов.

1.5. Выбор технологии для построения визуализатора

Визуализатор алгоритма – это программа (приложение), написанная в рамках определенной технологии. Как отмечалось выше, при использовании визуализаторов на лекциях, а также при заочном обучении возможно использование любых языков программирования и, соответственно, технологий. Обсуждение достоинств и недостатков различных подходов к построению приложений выходят за рамки данной работы. Отметим лишь, что в аудиторное преподавание позволяет писать на любом языке. При построении приложений, пригодных для использования в дистанционном обучении в Интернет-школе программирования требуется использование интернет-ориентированных технологий и языков.

При выборе технологической платформы для создания визуализаторов сформулируем требования, которым должна удовлетворять эта платформа:

1. Наличие готовых библиотек по построению пользовательского интерфейса.
2. Возможность встраивать приложения в *Web*-страницы.
3. Кросс-платформенность.
4. Возможность работы без использования браузера.
5. Широкое распространение и поддержка.

Существуют различные технологии для реализации приложений в Интернет. Примерами могут служить *Adobe Flash* [45], *Microsoft Silverlight* [46], *JavaScript* [47]. Несомненными достоинствами этих технологий является их широкое распространение и упрощенное написание Интернет-приложений. Их недостатком является их узкая область применения – они работают только в рамках браузеров. Несмотря на недавний выход в свет технологии *Adobe AIR* [48], позволяющей запускать *Flash*-приложения без браузеров, сама технология существенно уступает по гибкости и распространенности технологии *Java* [21]. Технология *Java Applets* [49], являющаяся частью технологической платформы *Java*, предоставляет максимальные преимущества по сравнению с другими языками и технологиями.

Приведем сравнительную таблицу потенциальных решений при выборе технологии построения визуализаторов (табл. 2), в которой цифрами 1 – 5 обозначены требования, указанные выше.

Таблица 2. Результаты сравнения платформ для разработки визуализаторов

Платформа	1	2	3	4	5
Flash	+	+	+	−	+
Adobe AIR	+	+	+	+	−
Silverlight	+	+	−	−	−
Delphi	+	−	−	+	−
Java Applets	+	+	+	+	+
JavaScript	±	+	+	±	±

Из результатов сравнения следует, что только платформа *Java* совместно с использованием технологии *Java Applets* дает все необходимые преимущества. Опишем эти преимущества более подробно.

- Интернет-ориентрированная технология – позволяет встраивать приложения в *Web*-сайты;
- *Java Applets* – это часть технологии *Java*. Это позволяет писать универсальные приложения для *Web* и одновременно создавать обычные приложения;
- *Java* является одной из самых распространенных технологий, поэтому использование этой технологии учащимися дает опыт использования профессиональных средств разработки, которые обязательно пригодятся в дальнейшей профессиональной деятельности.

При построении визуализаторов алгоритмов ручным способом, рассматриваемым в данной работе, целесообразно использовать процедурный язык. Первые визуализаторы писались с применением пакета *Delphi*, основанного на языке *Object Pascal*, который, однако, использовался лишь для пользовательского интерфейса (*VCL*). При этом вся логика требовала лишь процедурного подхода.

Автоматический подход к построению визуализаторов, изложенный в диссертации Г.А. Корнеева [43], требует использования объектно-ориентированного программирования. Необходимость этого обусловлена методом реализации рекурсивных алгоритмов на основе автоматного программирования [50]. Суть метода заключается в формировании стека автоматов, реализующих рекурсивные функции.

Из изложенного можно сделать вывод, что выбор *Java* в качестве технологии программирования и платформы для создания визуализаторов является целесообразным.

1.6. Автоматное программирование

В 1991 г. в России появилось *автоматное программирование* [44]. До появления автоматного программирования автоматы использовались в программировании, начиная с шестидесятых годов от случая к случаю. Только при построении компиляторов и протоколов они применялись систематически [51]. Автоматное программирование предлагает использовать автоматы гораздо шире: парадигма автоматного программирования предполагает использование автоматов для моделирования и реализации произвольной логики поведения сложных программ.

Автоматное программирование (называемое также программированием с явным выделением состояний) позволяет решать класс задач, связанный с системами управления со сложным поведением. Особенностью автоматов является наличие *состояний* в них. Автоматное программирование целесообразно использовать в задачах управления, в особенности технологическими процессами, например, в образовании. В частности, использование автоматного программирования удобно применять совместно с паттерном *Модель – Вид – Контроллер* [52, 53] при реализации контроллера.

Автором диссертационной работы предлагается использовать метод автоматного программирования для построения визуализаторов. Данная работа посвящена разработке и внедрению на практике этого метода.

Наличие состояний в автоматах позволяет выделять в алгоритме, реализуемым автоматом, контрольные точки, что, в свою очередь, позволяет преобразовывать алгоритм в набор статических кадров. Такой подход обеспечивает возможность однозначного и формализованного перехода от автомата, реализующего алгоритм, к визуализатору. При этом простая плавная анимация также являет собой последовательность быстро сменяющих друг друга кадров. При реализации такой анимации достаточно иметь систему автоматов, состояния которой однозначно преобразовываются в последовательность кадров.

1.7. Задачи, решаемые в диссертационной работе

Проблема построения визуализаторов алгоритмов состоит из двух частей. Первая из них – это технологии и методы, на основе которых должны строиться визуализаторы. Эта задача является инженерной и выходит за рамки настоящей диссертационной работы. Вторая часть, научная – это создание методов построения визуализаторов. При этом методы построения визуализаторов должны удовлетворять следующим критериям:

1. В разработанных методах должен присутствовать методический аспект. Учащийся, использующий методы построения визуализатора, должен обучаться как алгоритмам, которые он визуализирует, так и процессу построения приложений.
2. Методы должны быть универсальными и обеспечивать возможность построения визуализаторов любого алгоритма. При этом визуализаторы алгоритмов должны удовлетворять базовым требованиям к визуализаторам.

Приведем общие требования к визуализаторам алгоритмов, построенных разработанными методами:

1. Отображение входных и выходных данных в наглядной форме.
2. Наглядное отображение внутренних переменных алгоритма.
3. Отображение процесса воздействия алгоритма на данные.

4. Комментарии к визуализируемому действию алгоритма.
5. Отображение работы алгоритма по шагам.
6. Отображение работы алгоритма на различных уровнях.
7. Отображение простой анимации (плавных динамически-меняющихся иллюстраций).

Для построения указанных выше методов построения визуализаторов алгоритмов требуется решить следующие задачи:

1. Исследование возможности использования автоматного программирования для построения визуализаторов.
2. Разработка формализованного метода построения визуализаторов алгоритмов на основе конечных автоматов.
3. Разработка метода построения визуализаторов алгоритмов на основе системы взаимодействующих автоматов.
4. Разработка метода простой анимации алгоритмов при построении визуализаторов.
5. Проведение анализа предложенных методов построения визуализаторов и внедрение результатов работы в учебный процесс.

В рамках данной диссертации не рассматриваются методы для построения визуализаторов алгоритмов, использующих рекурсию. Исследования автора в этой области приведены в статье [74].

Выводы по главе 1

1. Анализ процесса обучения программированию и дискретной математике показал, что для эффективного обучения необходимо использовать визуализаторы.
2. Анализ существующих методов разработки и систем визуализации показал, что известные методы не позволяют формально преобразовывать логику алгоритма в логику построения визуализаторов. Таким образом, существует необходимость создания методов формализованного построения визуализаторов.

3. Ручной метод построения визуализаторов, несмотря на свою трудоемкость, является более эффективным инструментом при обучении программированию, чем полностью автоматизированный подход.
4. Традиционный эвристический подход к построению визуализаторов не может быть использован, как метод в силу отсутствия повторяемости и универсальности.
5. Методы автоматного программирования дают возможность явного выделения состояний в императивных алгоритмах, что дает возможность построить методы формализованного перехода от алгоритма к визуализаторам.
6. В качестве технологической платформы для построения визуализаторов выбрана технология *Java Applets*, как наиболее полно удовлетворяющая необходимым требованиям.
7. Сформулированы теоретические и практические задачи, решаемые в диссертационной работе.

ГЛАВА 2. МЕТОД ПОСТРОЕНИЯ ВИЗУАЛИЗАТОРОВ АЛГОРИТМОВ ДИСКРЕТНОЙ МАТЕМАТИКИ

Основным требованием к визуализатору является возможность пошаговой реализации алгоритма. Поэтому предлагаемый метод должен преобразовать исходный алгоритм в набор шагов для отображения на экране. Метод должен обеспечивать наглядность воздействия алгоритма на входные и служебные данные.

Ниже приводится основная идея, положенная в основу настоящей диссертации.

Прежде, чем излагать суть идеи остановимся на постановке задачи.

В рамках данной работы визуализатор представляет собой *динамический набор статических иллюстраций*, отображающих воздействие алгоритма на входные данные и служебные переменные. Целью построения визуализатора является формирование набора таких статических иллюстраций по ходу выполнения алгоритма. Таким образом, основная сложность при построении визуализатора состоит в преобразовании обычного императивного алгоритма в *набор контрольных точек* (состояний) системы. Наличие такого набора позволяет сформировать набор искомых иллюстраций.

Другим немаловажным аспектом при реализации визуализатора является возможность перехода от одной контрольной точки (иллюстрации) к следующей контрольной точке. В первой главе показано, что при традиционном эвристическом методе это достигалось путем разработки индивидуальных решений для каждого конкретного алгоритма. Наиболее универсальным способом являлось хранение полной истории состояний системы, что неэффективно в случае сложных алгоритмов обрабатывающих большие объемы данных.

Как уже отмечалось выше, программирование с явным выделением состояний предоставляет инструмент для формирования необходимых шагов.

Наличие *состояний* в полученной автоматной реализации алгоритма создает набор контрольных точек, в которых появляется возможность сформировать искомый *набор статических иллюстраций*. Поскольку автоматное решение реализует визуализируемый алгоритм – воздействует на выходные и выходные переменные таким же образом как императивный алгоритм, описанный в литературе, то для целей визуализации алгоритма поведение визуализатора, построенного на основе автоматной реализации алгоритма, не будет отличаться от воздействия реализации, построенной традиционным путем.

Поскольку при автоматной реализации алгоритма в каждом состоянии существует однозначный переход в следующее состояние, автоматный подход к реализации визуализаторов решает и вторую задачу: предоставляет возможность реализации переходов между контрольными точками алгоритма – переход от одной статической иллюстрации до другой получается автоматически без приложения специальных творческих усилий.

Из изложенного следует, что метод построения визуализаторов должен предоставить универсальный способ преобразования императивного алгоритма в автоматную реализацию алгоритма и включать в себя все необходимые этапы по переходу от словесной формулировки алгоритма к программной реализации визуализатора.

Визуализатор «по-крупному» предлагается строить следующим образом:

- по алгоритму строится автомат логики визуализатора (Контроллер – *Controller*);
- выбираются визуализируемые переменные (Модель – *Model*);
- проектируется формирователь иллюстраций и комментариев, который преобразует номер состояния и соответствующие значения визуализируемых переменных в «картинку» и поясняющий текст (Представление – *View*).

Такая конструкция визуализатора соответствует одному из основных паттернов проектирования объектно-ориентированных программ, нацеленных на взаимодействие с пользователем персонального компьютера, который

обозначается аббревиатурой *MVC* (*Model – View – Controller*) [52]. Особенностью этой технологии является четкое разделение логики поведения программы от ее визуального представления. В применении к визуализаторам алгоритмов использование паттерна *MVC* позволяет создавать вариации визуализаторов с единой логикой поведения, но различными технологиями реализации пользовательского интерфейса. Так, например, в одном случае интерфейс можно сделать с использованием технологии *Swing* [54] (именно этот способ рассматривается как основной в настоящей работе), в другом случае – *Java Applets* [49], а в третьем – с использованием технологии *AJAX* [55]. Наличие единой модели и логики в этом случае будут гарантировать единообразие в поведении визуализатора в разных пользовательских окружениях.

В разд. 2.1 рассматривается эвристический автоматный подход к построению визуализаторов. В основе этого подхода лежит уже сформулированная идея, однако он обладает указанным выше недостатком – построение автоматной реализации алгоритма требует специальных навыков, которые не всегда применимы. Поэтому этот подход не может называться методом.

В разд. 2.2 рассматриваются две модификации базового метода построения визуализаторов на основе автоматов Мили и автоматов Мура, происходит сравнение модификаций.

В разд. 2.3 приводится сравнение эвристического, автоматного эвристического подходов с формализованным методом построения визуализаторов алгоритмов.

2.1. Автоматный эвристический подход

Основным недостатком при построении визуализаторов является то, что обычно применяются только такие понятия, как «входные и выходные переменные», а понятие «состояние» в явном виде не используется. При традиционном подходе состояния задаются неявно, как значения внутренних

переменных, которые часто называются «флагами». Однако, по мнению автора, совершенно естественным кажется, что каждый шаг визуализации необходимо проводить в соответствующем явно выделенном состоянии или на переходе. Поэтому при построении программ визуализации целесообразно использовать технологию автоматного программирования, базирующуюся на конечных автоматах [44].

Простейший подход к использованию автоматов для построения визуализаторов, который, видимо, применим только для очень простых алгоритмов, состоит в непосредственном построении графа переходов по словесному описанию алгоритма.

Основным требованием к визуализатору является возможность пошаговой реализации алгоритма. Поэтому предлагаемый метод должен преобразовать исходный алгоритм в набор шагов для отображения на экране.

В основе автоматного эвристического подхода лежит построение автоматной реализации алгоритма с целью выделения состояний и последующей визуализации. После предложения идеи использования автоматного подхода к построению визуализаторов в 2001 г. первое, что было апробировано это *автоматный эвристический подход*. Целью этого подхода являлось построение автомата, описывающего логику поведения визуализатора, реализующего алгоритм. Однако поскольку формализованный метод отсутствовал, то такой подход подходил для очень малого числа алгоритмов. Для более сложных алгоритмов явное выделение состояний было весьма трудоемкой задачей.

Однако, поскольку в основе формализованного метода лежит именно этот подход, остановимся на демонстрации использования этого подхода для реализации простейшего визуализатора. При автоматном эвристическом подходе формируются основные шаги построения визуализатора.

1. Постановка и решение задачи в словесной форме.
2. Автоматная реализация алгоритма решения задачи.
3. Реализация построителя иллюстраций.

4. Реализация визуализатора.

Рассмотрим применение этого подхода к простой задаче из теории алгоритмов – «Решето Эратосфена» [56]. Этот классический алгоритм, хоть и не является широко распространенным в силу своей малой эффективности, но является достаточно полезным при изучении дискретной математики.

1. Постановка задачи и решение задачи

Решето Эратосфена — алгоритм нахождения всех простых чисел до некоторого целого числа *n*, который приписывают древнегреческому математику Эратосфену [57].

Приведем словесное описание усовершенствованного алгоритма решения этой задачи.

Для нахождения всех простых чисел не больше заданного числа n, следуя методу Эратосфена, необходимо выполнить следующие шаги:

1. *Выписать подряд все целые числа от двух до n.*
2. *Пусть переменная p изначально равна двум — первому простому числу.*
3. *Вычеркнуть из списка все числа от p^2 до n, делящиеся на p.*
4. *Найти первое, не вычеркнутое число, большее, чем p, и присвоить значению переменной p это число.*
5. *Повторять шаги 3 и 4 до тех пор, пока p^2 не станет больше, чем n.*
6. *Все не вычеркнутые числа в списке — простые числа.*

2. Автоматная реализация решения задачи

Приведенный алгоритм можно реализовать при помощи автомата, приведенного на рис. 1. При этом предполагается, что $a[i]$ – булево значение, обозначающее, что данное число является простым.

Приведенный автомат реализует алгоритм «Решето Эратосфена». Далее в соответствии с автоматным подходом требуется принять решение о формировании иллюстраций. Поскольку формализованный метод отсутствует, то решение должно быть также принято «из общих соображений».

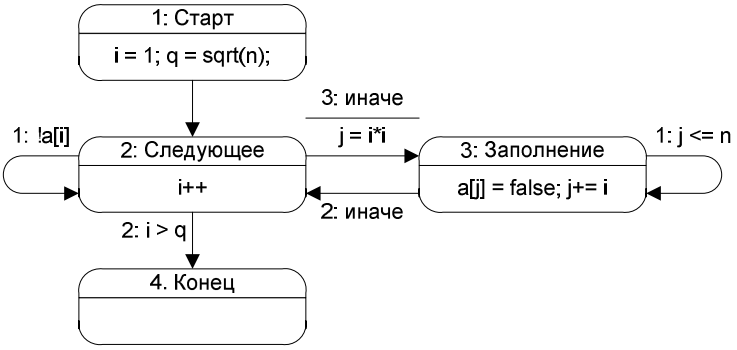

Рис. 1. Смешанный автомат алгоритма «Решето Эратосфена»

3. Реализация построителя иллюстраций

Отметим, что рассмотрев граф переходов автомата (рис. 1), можно заключить, что удобно показывать следующие иллюстрации:

1. Начальное состояние – отображается в состоянии 1.
2. Поиск уже известных чисел (шаг 2 и 4 словесного алгоритма) – отображается в состоянии 2.
3. Вычеркивание непростых чисел – в состоянии 3.
4. Отображение финального результата – в состоянии 4.

Отметим, что даже на таком простом алгоритме видна несостоятельность эвристического подхода. Отметим недостатки графа переходов, представленного на рис. 1.

1. Поскольку приоритет переходов в состоянии 2 расположен в соответствии с формулировкой задачи, то визуализатор будет проходить по лишним числам. Например, в случае, если $n = 50$, то последнее рассмотренное число должно быть 8, в то время как визуализатор покажет 11 (первое простое число большее $\sqrt{n}$). При формализованном методе построения визуализатора, рассмотренном ниже, этот недостаток устраняется путем отладки императивного алгоритма, написанного на языке программирования. Безусловно, отладку можно производить и на автоматах, но эта задача требует более высокого уровня подготовки обучающихся.

2. В состоянии 3, как можно отметить, значение служебной переменной *j* изменяется сразу же после изменения флага *a*[*j*]. Таким образом, при отображении иллюстраций требуется вводить специальную логику для анализа ситуации в состоянии 3.

С учетом указанных недостатков, автомат (рис. 1) предлагается трансформировать в другой автомат. Этот автомат не является эквивалентным в силу того, что он проходит меньший интервал значений *i*, но является более правильным и простым с точки зрения визуализации, поскольку устраняет указанные недостатки (рис. 2).

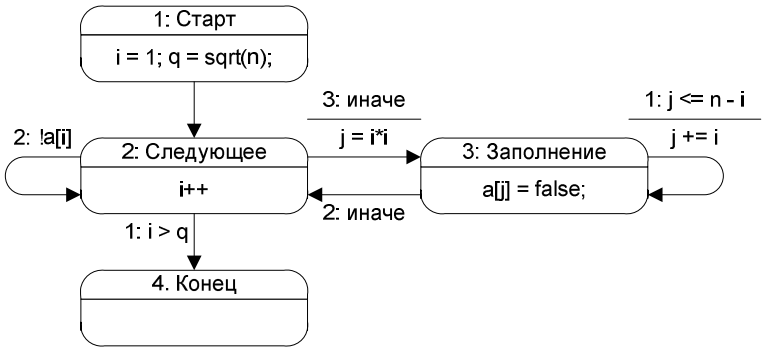

Рис. 2. Усовершенствованный автомат алгоритма «Решето Эратосфена»

В усовершенствованном автомате формирование иллюстраций происходит простым способом – каждое состояние формирует отдельную иллюстрацию.

Состояние 1 показывает исходный массив чисел (рис. 3).

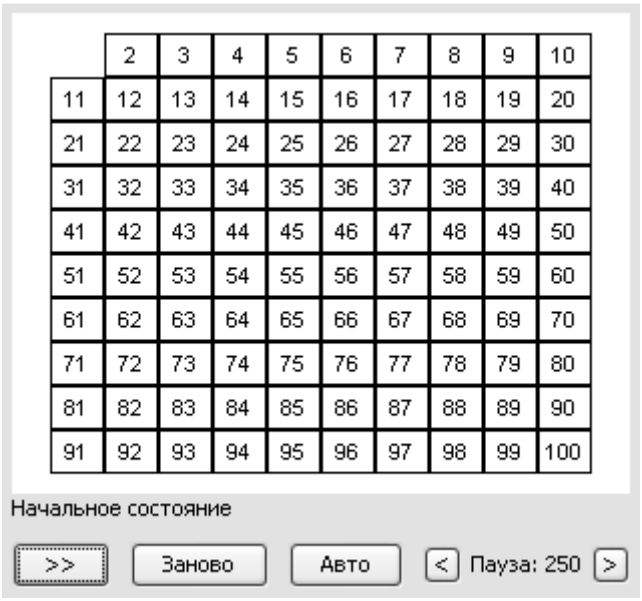

Рис. 3. Состояние 1 – начальное состояние

В **состоянии 2** осуществляется поиск следующего простого числа (рис. 4).

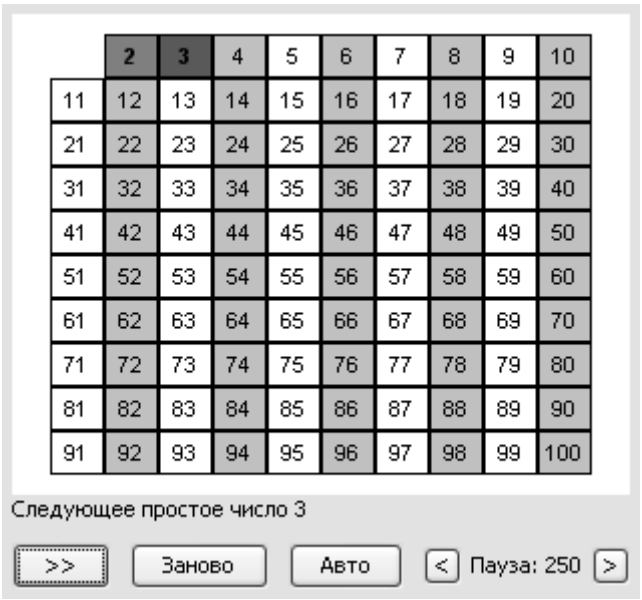

Рис. 4. Состояние 2 – поиск следующего простого числа

В **состоянии 3** осуществляется вычеркивание непростых чисел (рис. 5).

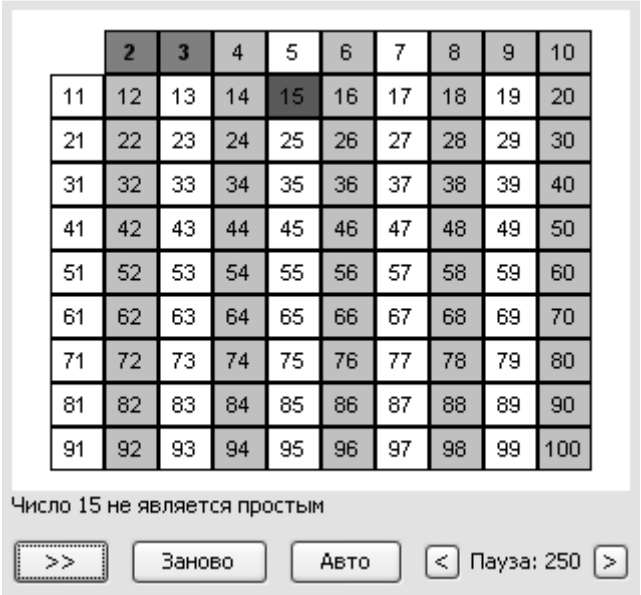

Рис. 5. Состояние 3 – вычеркивание непростых чисел

Состояние 4 является конечным и отображает все не вычеркнутые числа как простые (рис. 6).

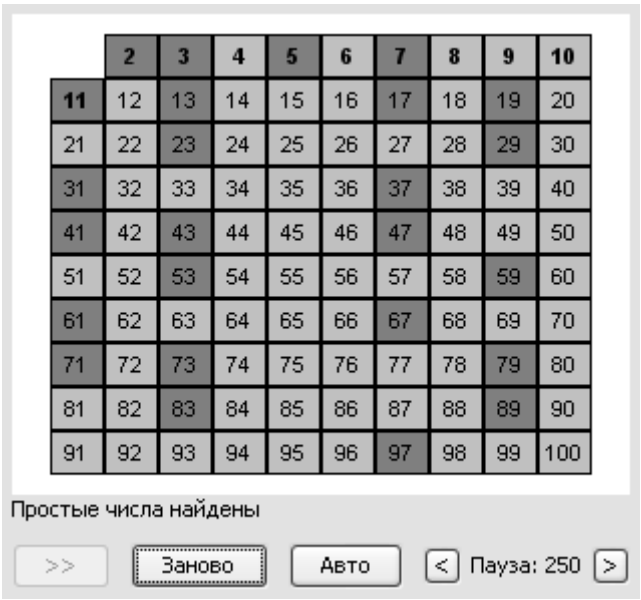

Рис. 6. Конечное состояние – простые числа найдены

4. Реализация визуализатора

Процесс реализации визуализатора не представляет особого интереса в рамках данного раздела. Отметим лишь, что после формирования автомата и иллюстраций, оставшиеся этапы построения являются техническими. Более детально они будут рассмотрены в следующем разделе при изложении формализованного метода построения визуализаторов алгоритмов.

Из изложенного следует, что несмотря на то, что эвристический автоматный подход может использоваться для построения некоторых визуализаторов, методом он по своей сути не является, поскольку обладает следующими недостатками.

1. Построение автоматной реализации алгоритма требует специальных навыков. В некоторых случаях решение этой задачи может потребовать существенных трудозатрат.
2. При построении автоматного решения задачи, автомат не всегда является готовым к использованию в визуализаторе.

Следует, однако, отметить, что в том случае, когда автоматное решение было найдено и визуализатор простроен, автомат, реализующий визуализируемый алгоритм, может содержать меньшее число состояний. Так при формализованном подходе с использованием автоматов Мили, изложенном в разд. 2.2.2, автомат имел бы пять состояний, а при формализованном подходе с использованием автомата Мура, изложенном в разд. 2.2.3, – шесть.

2.2. Метод построения визуализаторов на основе автоматного подхода

После рассмотрения эвристического подхода перейдем к формализованному подходу. Как было установлено в предыдущем разделе, основным недостатком эвристического подхода является отсутствие формализованного метода перехода от словесной формулировки к автомату, реализующему логику поведения визуализатора.

Существует две модификации метода разработки визуализаторов алгоритмов ручным способом. Оба метода основаны на использовании программирования с явным выделением состояний и применении *SWITCH*-технологии [59].

Первая модификация метода основана на реализации поведения визуализатора алгоритма с помощью автомата Мили. При использовании этой модификации управляющие состояния не содержат действий над

управляемыми переменными, в то время как все действия осуществляются на переходах.

Вторая модификация метода основана на реализации поведения визуализатора с помощью автомата Мура. При использовании этой модификации действия совершаются в управляющих состояниях.

Приведем этапы метода формализованного построения визуализаторов алгоритмов, который назовем *базовым*:

1. **Постановка задачи**.
2. **Решение задачи** (в словесно-математической форме).
3. **Выбор визуализируемых переменных**.
4. **Анализ алгоритма для визуализации**. Анализируется решение с целью определения того, что и как отображать на экране.
5. **Реализация алгоритма решения задачи**.
6. **Реализация алгоритма на выбранном языке программирования**. На этом шаге производится реализация алгоритма, его отладка и проверка работоспособности.
7. **Построение схемы алгоритма по программе**.
8. **Преобразование схемы алгоритма в граф переходов автомата Мили либо Мура**.
9. **Формирование набора невизуализируемых переходов (состояний)**.
10. **Выбор интерфейса визуализатора**.
11. **Сопоставление иллюстраций и комментариев с состояниями автомата**.
12. **Выбор архитектуры программы визуализатора**.
13. **Программная реализация визуализатора**.

Известно, что обычно создание программы разбивается на пять стадий: разработка требований, анализ, проектирование, реализация и тестирование. В данном случае к разработке требований относятся первые три пункта технологии. Анализ – пункты 4 и 5. К проектированию относятся пункты

с 6-го по 11-й. В отличие от процедурного подхода для объектно-ориентированного подхода, выбор архитектуры программы относится не к стадии проектирования, а к стадии реализации, так как все предыдущие пункты метода не зависят от реализации. Естественно, что к стадии реализации относится и последний пункт метода.

Как будет показано ниже, при использовании предлагаемого метода тестирование резко упрощается, а время написания визуализатора сокращается.

Продемонстрируем предложенный метод на примере построения визуализатора алгоритма, использующего динамическое программирование [2], которое является одним из самых сложных разделов теории алгоритмов.

Ниже приведены два варианта реализации «дискретной целочисленной задачи о рюкзаке» – на основе автоматов Мили и автоматов Мура.

2.2.1. Общая часть метода построения визуализаторов на основе автоматов Мили и Мура

Отметим, что этапы метода с 1-го по 7-ой не зависят от выбора разновидности метода, в то время как этапы с 8-го по 13-й зависят от того, какие используются автоматы – автоматы Мили или Мура. Поэтому первые семь этапов сделаем общими для двух модификаций.

1. Постановка задачи

Рассмотрим словесную постановку задачи, которая в литературе носит название «дискретная целочисленная задача о рюкзаке» [1, 4]. Приведем одну из формулировок этой задачи.

Имеется набор из K неделимых предметов, для каждого из которых известна масса M_i (в кг), являющаяся натуральным числом. Задача состоит в том, чтобы определить, существует ли хотя бы один набор предметов, размещаемый в рюкзаке, суммарная масса которых равна N. Если такой набор существует, то требуется определить его состав.

2. Решение задачи

Приведем словесную формулировку решения этой задачи.

1. Построение функции, представляющей оптимальное решение.

Построим вектор $M(1..K)$, каждый элемент M_i которого соответствует массе i-го предмета.

Введем функцию $T(i, j)$, значение которой равно единице, если среди предметов с первого по i-ый существует хотя бы один набор предметов, сумма масс которых равна j, и нулю – в противном случае. Выходной массив *positions* будет содержать искомый набор переменных.

2. Рекуррентные соотношения, связывающие оптимальные значения для подзадач.

Определим начальные значения функции T:

- $T(0, j) = 0$ при $j \geq 1$ {без предметов нельзя набрать массу $j \geq 1$};
- $T(i, 0) = 1$ при $i \geq 0$ {всегда можно набрать нулевую массу}.

Определим возможные значения функции T при других значениях аргументов.

Существуют две возможности при выборе предметов: включать предмет с номером i в набор или не включать. Это в математической форме может быть записано следующим образом:

- если предмет с номером i не включается в набор, то решение задачи с i предметами сводится к решению подзадачи с $(i - 1)$-им предметом. При этом $T(i, j) = T(i - 1, j)$;
- если предмет с номером i включается в набор, то это уменьшает суммарную массу для $i - 1$ первых предметов на величину M_i. При этом $T(i, j) = T(i - 1, j - M_i)$. Эта ситуация возможна только тогда, когда $M_i \leq j$.

Для получения решения необходимо выбрать большее из значений рассматриваемой функции. Поэтому рекуррентные соотношения при $i \geq 1$ и $j \geq 1$ имеют вид:

- $T(i, j) = \mathrm{T}(i - 1, j)$ при $j < M_i$;
- $T(i, j) = \max\left(T(i - 1, j), T(i - 1, j - M_i)\right)$ при $j \geq M_i$.

После построения таблицы T проверяется элемент $T(K, N)$. Если этот элемент равен единице, то искомый набор может быть получен.

3. Задание исходных данных.

- Пусть $K = 5$ – заданы пять предметов.
- Эти предметы имеют следующие массы: $M_1 = 4$; $M_2 = 5$; $M_3 = 3$; $M_4 = 7$; $M_5 = 6$.
- Суммарная масса предметов, которые должны быть размещены в рюкзаке, равна 16 ($N = 16$).

4. Вычисление значений функции *T (i, j)* (прямой ход алгоритма).

Используя приведенные выше рекуррентные соотношения, построим таблицу значений рассматриваемой функции (табл. 3).

Таблица 3. Значения рекуррентной функции

$T(i,j)$		J																
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	2	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
	3	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0
	4	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1
	5	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Из таблицы следует, что $T(5, 16) = 1$, и поэтому существует хотя бы один набор предметов, удовлетворяющий постановке задачи.

5. Определение набора предметов (обратный ход алгоритма).

Во время обратного хода алгоритма происходит поиск искомого набора, начиная с ячейки $T(K, N)$. При этом переход с ячейки $T(i, j)$ в следующую ячейку происходит следующим образом:

- Если значение $T(i - 1, \text{j})$ равно нулю, то не существует набора среди элементов с первого по $i - 1$-й, суммарный вес которых равен j. Поэтому предмет с номером i входит в искомый набор, а следующей рассматриваемой ячейкой становится $T(i - 1, j)$.

- В случае, когда значение $T(i-1, \text{j})$ равно 1, то следующей рассматриваемой ячейкой становится $T(i-1, \text{j})$.
- Обратный ход алгоритма происходит до тех пор, пока значение i не станет равным нулю.

Разберем подробнее действие обратного хода алгоритма в рассматриваемом примере. Рассмотрим элементы $T(5, 16)$ и $T(4, 16)$ таблицы. Так как их значения равны, то массу в 16 кг можно набрать с помощью первых четырех предметов – пятый предмет в набор можно не включать.

Теперь рассмотрим элементы $T(4, 16)$ и $T(3, 16)$. Их значения не равны, и поэтому набор из трех предметов не обеспечивает решение задачи. Следовательно, четвертый предмет должен быть включен в набор. Поэтому «остаток» массы, размещаемой в рюкзаке, который должен быть набран из оставшихся предметов, равен: $16 - M_4 = 16 - 7 = 9$.

Для элементов $T(3, 9)$ и $T(2, 9)$ значения равны – третий предмет в набор не включается.

Для элементов $T(2, 9)$ и $T(1, 9)$ значения не равны – второй предмет должен быть включен в набор. При этом остаток равен: $9 - M_2 = 9 - 5 = 4$.

И, наконец, рассмотрим элементы $T(1, 4)$ и $T(0, 4)$. Их значения не равны – первый предмет должен быть включен в набор, а «остаток» массы для других предметов равен нулю.

Таким образом, одно из решений задачи найдено – в набор включены первый, второй и четвертый предметы.

3. Выбор визуализируемых переменных

Перечислим переменные, которые содержат значения, необходимые для визуализации:

- i – номер предмета;
- j – сумма весов предметов;
- $T[][]$ – значения функции T;
- *positions* – номера искомых предметов.

4. Анализ алгоритма для его визуализации

Из рассмотрения алгоритма следует, что его визуализация должна выполняться следующим образом:

- показать таблицу, заполненную начальными значениями;
- отразить прямой ход решения задачи – процесс построения таблицы значений $T(i, j)$;
- отразить обратный ход решения задачи – процесс поиска набора предметов за счет «обратного» движения по указанной таблице.

5. Реализация алгоритма решения задачи

Приведем алгоритм решения задачи на псевдокоде [2]:

```
1    for j ← 1 to N
2      do T [0, j] ← 0
3    for i ← 0 to K
4      do T [i, 0] ← 1
5    for i ← 1 to K
6      do for j ← 1 to N
7        do if j < M [i]
8          then T[i, j] ← T[i – 1, j]
9          else T[i, j] ← max (T[i – 1, j],
10                              T[i – 1, j – M[i]])
11   j ← N
12   if T[K, j] = 1
13     then  for i ← K downto 1
14             do if T[i – 1, j] = 0
15                   then  добавить в искомый набор
16                         j ← j – M[i]
17           return искомый набор
18     else  return «набор не существует»
```

6. Реализация алгоритма на языке *Java*

Перепишем программу с псевдокода на язык *Java* (листинг 1).

Листинг 1. Текст программы, реализующей решение «задачи о рюкзаке»

```
1  /**
```

```
2    * @param N суммарный вес предметов в рюкзаке
3    * @param M массив весов предметов
4    * @return список номеров предметов, сумма весов которых
5    *         равна N, либо null, если это невозможно
6    */
7   private static Integer[] solve(int N, int[] M) {
8     // Переменные циклов
9     int i, j;
10    // Число предметов
11    int K = M.length;
12    // Массив T
13    int[][] T = new int[K + 1][N + 1];
14    // Результирующие номера
15    Collection<Integer> positions = new ArrayList<Integer>();
16    // Результат работы true, если суммарный вес можно получить
17    boolean result = false;
18
19    // Определение начальных значений функции T
20    for (j = 1; j <= N; j++) {
21      T[0][j] = 0;
22    }
23    for (i = 0; i <= K; i++) {
24      T[i][0] = 1;
25    }
26
27    // Построение таблицы значений функции T
28    for (i = 1; i <= K; i++) {
29      for (j = 1; j <= N; j++) {
30        if (j < M[i - 1]) {
31          T[i][j] = T[i - 1][j];
32        } else {
33          T[i][j] = Math.max(T[i - 1][j], T[i - 1][j - M[i - 1]]);
34        }
35      }
36    }
37
38    // Определение набора предметов
39    j = N;
40    if (T[K][j] == 1) {
41      // Решение существует
42      for (i = K; i >= 1; i--) {
43        if (T[i][j] != T[i - 1][j]) {
44          positions.add(i);
45          j -= M[i - 1];
46        }
47      }
48      // Решение найдено
49      result = true;
50    }
51
52    // Если решение найдено, то оно возвращается,
53    // иначе возвращается null
54    return (Integer[])(result ? positions.toArray(new Integer[0]) : null);
55  }
```

В этой программе операции ввода/вывода не приведены, так как их будет выполнять визуализатор.

7. Построение схемы алгоритма по программе

Построим по тексту программы схему алгоритма. Отметим, что, несмотря на то, что схема алгоритма строится одинаково независимо от

выбираемого типа автоматов, в соответствии с методом преобразования итеративных программ в автоматные [58] состояниями маркируются разные элементы схемы алгоритма. Таким образом, в зависимости от модификации метода, для которого строится схема, выбирается один из двух вариантов (рис. 7 или рис. 20).

2.2.2. Построение визуализаторов на основе автоматов Мили

В работе [58] показано, что при программной реализации по схеме алгоритма может быть построен как автомат Мура, так и автомат Мили. В автоматах первого типа действия выполняются в вершинах, а в автоматах второго типа – на переходах. Визуализаторы могут быть построены как с использованием автоматов Мура, так и автоматов Мили. Для уменьшения числа состояний в настоящей разновидности применяются автоматы Мили.

8. Преобразование схемы алгоритма в граф переходов автомата Мили

В соответствии с методом, изложенным в работе [58], определим состояния, которые будет иметь автомат Мили, соответствующий этой схеме алгоритма. Для этого присвоим номера точкам, следующим за последовательно расположенными операторными вершинами (последовательность может состоять и из одной вершины). Номера присваиваются также начальной и конечным вершинам.

Число состояний, определенных описанным методом, бывает для целей визуализации недостаточным, так как в ряде случаев необходимо показывать также условия и результаты некоторых действий.

Исходя из изложенного, для схемы алгоритма на Рис. 1 выделим девять состояний автомата с номерами 0 – 6, 9. Еще одно состояние (с номером 7) введено для визуализации обратного хода алгоритма. И, наконец, состояние с номером 8 используется для визуализации результата выбора на обратном ходе. Таким образом, автомат визуализации будет иметь одиннадцать состояний, а

его граф переходов, который строится за счет определения путей между выделенными точками на схеме алгоритма – одиннадцать вершин (рис. 7).

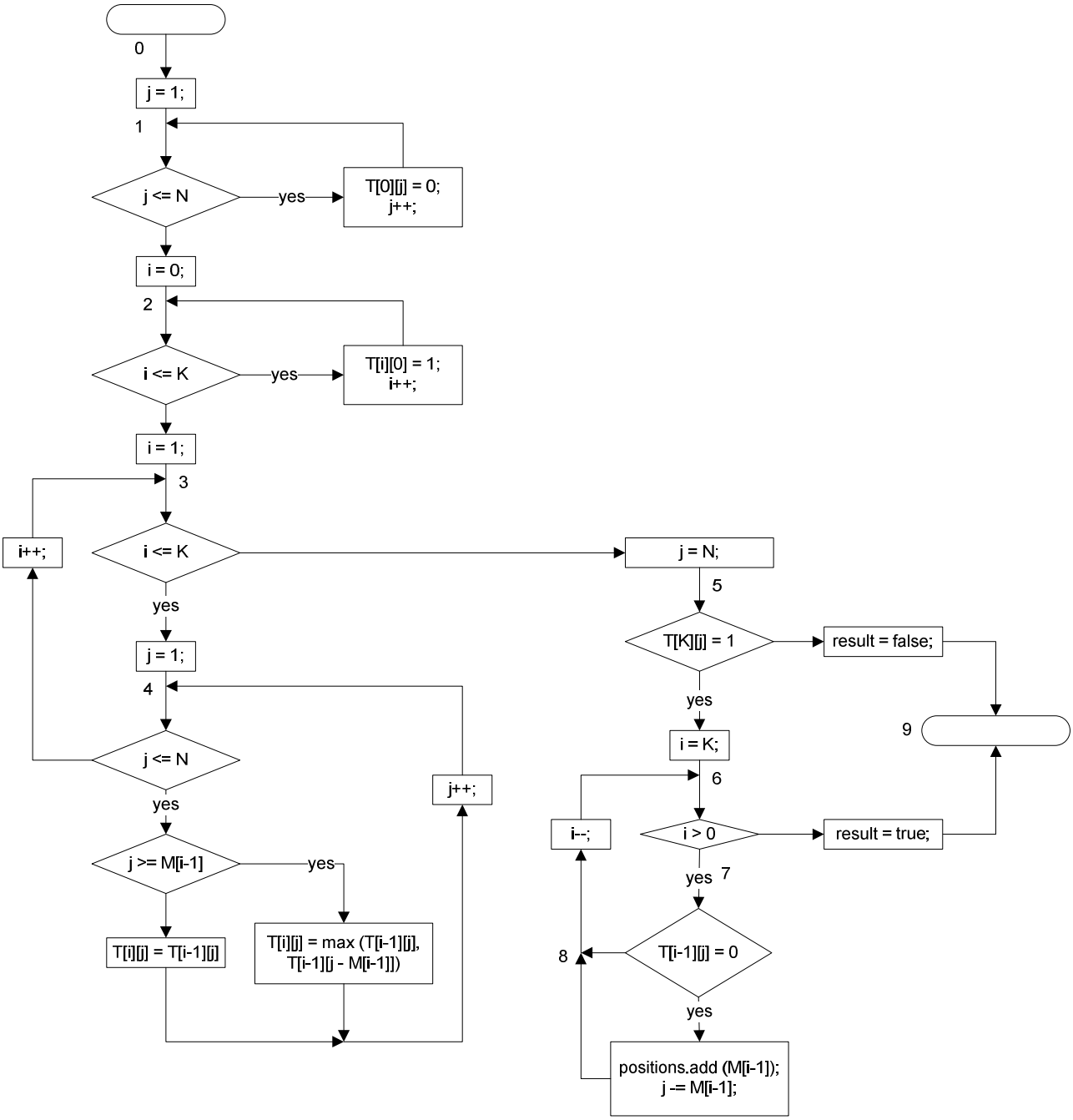

Рис. 7. Схема алгоритма решения задачи о рюкзаке

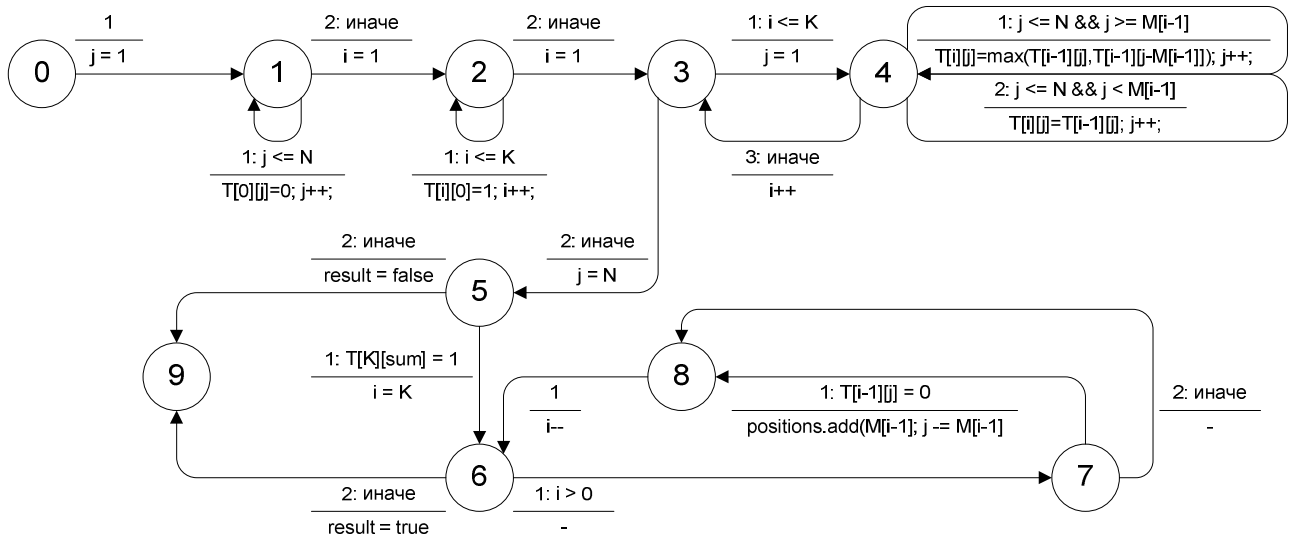

Рис. 8. Граф переходов для визуализации логики алгоритма

Как следует из графа переходов, на петлях в состояниях 1 и 2 осуществляется заполнение таблицы значений функции *T* начальными данными в первой строке и в первом столбце. На петлях в состоянии 4 реализуется алгоритм заполнения таблицы значений функции *T*. Переходы из состояния 7 в состояние 8 отображают первую половину шага алгоритма обратного хода, причем более сложный переход соответствует включению предмета с номером *i* в результирующий набор, а второй переход – не включению предмета в этот набор. Переход из состояния 8 в состояние 6 соответствует второй половине шага алгоритма обратного хода.

9. Формирование набора невизуализируемых переходов

Из анализа графа переходов (Рис. 2) следует, что «неинтересными» (не изменяющими визуализацию) для пользователя являются следующие переходы: 1 → 2, 3 → 4, 3 → 5, 4 → 3, 6 → 7, 6 → 9. Устранение таких переходов осуществляется за счет того, что один шаг работы визуализатора состоит из нескольких переходов автомата, реализующего его логику.

Для исключения «неинтересных» переходов шаг визуализатора реализуется следующим образом:

```
1  while следующий переход in (0 → 1, 1 → 2, 3 → 4, 3 → 5,
2                               4 → 3, 6 → 7, 6 → 9)
3    do сделать переход автомата
```

10.Выбор интерфейса визуализатора

В верхней части визуализатора (в дальнейшем называемой *иллюстрацией*) представляется следующая информация:

- таблица значений функции $T(i, j)$, где $i = 0, \ldots, 5$; $j = 0, \ldots, 16$;
- набор весов предметов (крайний левый столбец с цифрами 4, 5, 3, 7, 6).

В нижней части визуализатора расположена панель управления, которая содержит следующие кнопки:

- «>>» – сделать шаг алгоритма;
- «Заново» – начать алгоритм заново;
- «Авто» – перейти в автоматический режим;
- «<», «>» – изменение паузы между автоматическими шагами алгоритма.

Среднюю часть визуализатора занимает область комментариев, в которой на каждом шаге отображается номер состояния и описание действия, выполняемого в этом состоянии.

Визуализатор в исходном состоянии, которое соответствует начальному состоянию автомата, представлен на рис. 9.

В следующем разделе приведены иллюстрации, отображающие внешний вид визуализатора в различных состояниях, которые должны формироваться в процессе его работы. Они перенесены в статью с указанием соответствующих состояний, как скриншоты работающего визуализатора.

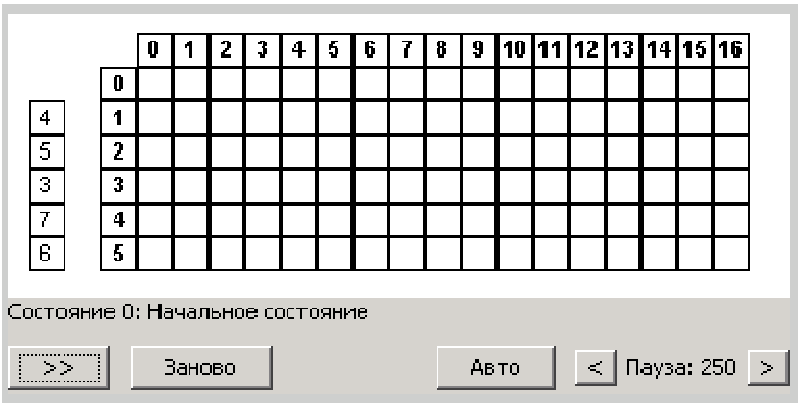

Рис. 9. Начальное состояние визуализатора

11.Сопоставление иллюстраций и комментариев с состояниями автомата

Ввиду того, что при построении визуализатора используется автомат Мили, то будем в рассматриваемом состоянии отображать **действия, которые будут выполнены при переходе** из него. При этом серым цветом подсвечивается текущая ячейка таблицы, а черным – рабочие ячейки. При этом на прямом ходе ищется значение в текущей ячейке, исходя из значений в рабочих ячейках. На обратном ходе – значение в текущей ячейке сравнивается со значениями в рабочих ячейках для определения вхождения элемента в набор.

Состояние 0. Описано в предыдущем разделе (рис. 9).

Состояние 1. Заполнение нулями верхней строки таблицы *T*. Обратим внимание, что одному управляющему состоянию визуализатора обычно соответствует некоторое число вычислительных состояний (шагов визуализации) [60]. Поэтому состояние представляется двумя иллюстрациями, первая из них (рис. 10) соответствует первому шагу в этом состоянии, а вторая (рис. 11) – последнему.

Состояние 2. Заполнение единицами левого столбца таблицы *T* (рис. 12).

Состояние 3 является «невизуализируемым», так как оно служебное.

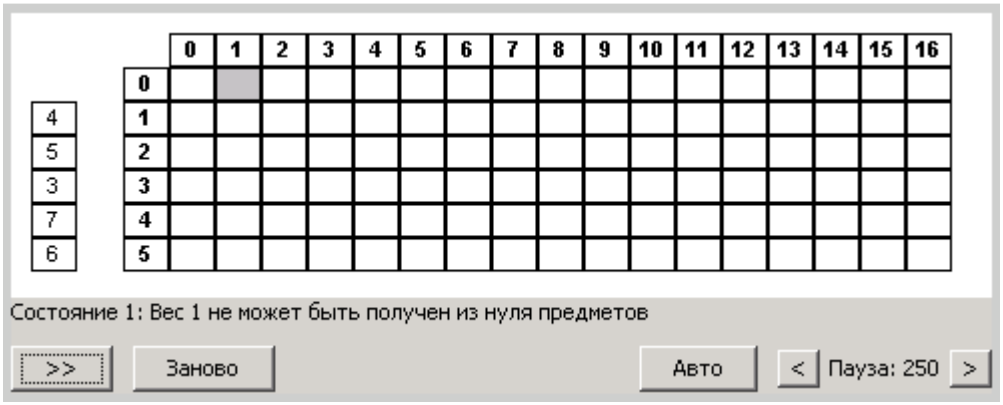

Рис. 10. Состояние 1. Начало

Рис. 11. Состояние 1. Конец

Рис. 12. Состояние 2

Состояние 4 может отображаться в двух различных вариантах, так как граф на рис. 8 имеет в этом состоянии две петли, помеченные разными условиями и соответствующими действиями. На рис. 13 приведена иллюстрация, соответствующая нижней петле, в которой выполняется копирование значения из расположенной выше ячейки таблицы (помечена темным цветом) в ячейку, расположенную ниже (помечена серым цветом), поскольку рассматриваемый в данный момент предмет не может быть

использован в наборе. Это имеет место в случае, когда сумма весов предметов для текущего столбца меньше веса предмета, соответствующего текущей строке. На рис. 14 приведена иллюстрация для верхней петли в рассматриваемом состоянии, на которой показано, что имеет место альтернатива в выборе значения, поскольку вес T (i, j) может быть получен двумя разными способами.

При использовании первого способа текущий предмет включается в набор, а для второго способа рассматриваемый предмет в набор не включается. Наличие единицы в одной из двух темных (рис. 14) ячеек соответствует одному из вариантов. Поскольку на рис. 14 выполняется соотношение $T(1, 4) = 1$, то набор с весом 4 может быть получен с применением одного предмета. Поэтому набор с весом 9 может быть получен из двух предметов за счет добавления второго предмета с весом 5. Следовательно, $T(2, 9) = 1$.

Рис. 13. Состояние 4 (нижняя петля)

Рис. 14. Состояние 4 (верхняя петля)

На этом прямой ход алгоритма завершен.

Состояние 5 отображает проверку того, что искомый суммарный вес набора предметов может быть получен, поскольку значение в правой нижней ячейке равно единице (рис. 15). Это состояние является промежуточным между прямым и обратным ходом алгоритма, на котором собственно и выполняется поиск результирующего набора.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	**1**	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
5	**2**	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
3	**3**	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0
7	**4**	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1
6	**5**	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Рис. 15. Состояние 5

Рассмотрим состояния автомата, соответствующие обратному ходу алгоритма.

Состояния 6 – 8 автомата, образующие замкнутый контур на рис. 8, порождают искомый набор предметов. Опишем, как эти состояния иллюстрируются.

Состояние 6 не изображается.

Состояние 7 отображает альтернативу первой половины шага при обратном ходе алгоритма (рис. 16) из серой ячейки в одну из черных. Вариант перехода в ячейку в том же столбце означает, что текущий предмет не включается в результирующий набор. Переход во вторую черную ячейку означает включение предмета с номером i в набор. При этом отметим, что собственно переход всегда осуществляется в ячейку, помеченную единицей. В случае если обе ячейки содержат по единице, то выбирается первый из описанных вариантов.

Состояние 8 отображает результат выбора на предыдущем шаге (рис. 17). Из левой части рисунка следует, что предмет с весом 7 включен в результирующий набор.

Рис. 16. Состояние 7

Рис. 17. Состояние 8

Состояние 9 отображает искомый результат – набор предметов, реализующих суммарный вес, равный 16 (рис. 18).

Рис. 18. Состояние 10

12.Выбор архитектуры программы визуализатора

Визуализатор предлагается реализовывать на основе паттерна *Модель – Вид – Контроллер*.

В данном случае *Модель* – визуализируемые переменные, которые являются глобальными (*Globals*), *Вид* – формирователь иллюстраций и комментариев (*Drawer*), а *Контроллер* – автомат (*Automaton*). Объединение указанных составляющих осуществляется с помощью четвертой компоненты – *Applet*, которая осуществляет также реализацию панели управления.

Диаграмма классов [61], реализующая этот паттерн для рассматриваемого примера, приведена на рис. 19.

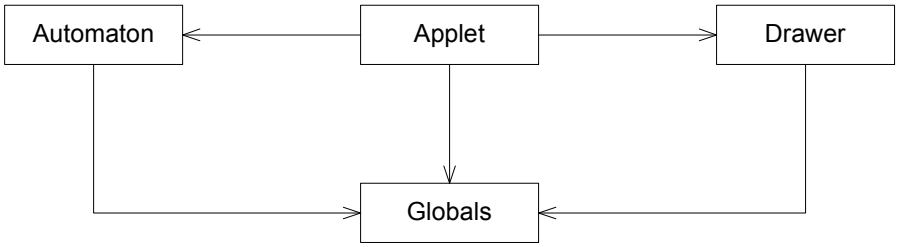

Рис. 19. Диаграмма классов визуализатора

В качестве глобальных переменных используются переменные, описанные на этапе 3. Спроектированный автомат описан на этапе 8, а иллюстрации и комментарии, привязанные к состояниям автомата – на этапах 10, 11.

13.Программная реализация визуализатора

В работе [59] было предложено переходить формально и изоморфно от графа переходов автомата к его программной реализации на основе оператора *switch*. Для рассматриваемого примера результатом преобразования графа переходов (рис. 8) является текст программы, приведенный ниже:

```
1           private void makeAutomationStep() {
2               switch (state) {
3                   case 0:     // Начальное состояние
4                       j = 1;
5                       state = 1;
6                       break;
7
8                   case 1:     // Заполнение 0-й строки
9                       if (j <= N) {
10                          T[0][j] = 0;
11                          j++;
12                      } else {
13                          i = 0;
14                          state = 2;
15                      }
16                      break;
17
```

```
18                  case 2:     // Заполнение 0-го столбца
19                      if (i <= K) {
20                          T[i][0] = 1;
21                          i++;
22                      } else {
23                          i = 1;
24                          state = 3;
25                      }
26                      break;
27
28                  case 3:     // Переход к следующей строке
29                      if (i <= K) {
30                          j = 1;
31                          state = 4;
32                      } else {
33                          sum = N;
34                          state = 5;
35                      }
36                      break;
37
38                  case 4:     // Заполнение очередной ячейки
39                      if (j <= N && j >= M[i - 1]) {
40                          T[i][j] = Math.max(T[i - 1][j], T[i - 1][j - M[i - 1]]);
41                          j++;
42                      } else if (j <= N && j < M[i - 1]) {
43                          T[i][j] = T[i - 1][j];
44                          j++;
45                      } else {
46                          i++;
47                          state = 3;
48                      }
49                      break;
50
51                  case 5:     // Конец заполнения таблицы
52                      if (T[K][sum] == 1) {   // Решение найдено
53                          i = K;
54                          state = 6;  // Делать обратный ход
55                      } else {
56                          result = false;
57                          state = 9;  // Обратный ход не требуется
58                      }
59                      break;
60
61                  case 6:     // Обратный ход
62                      if (i > 0) {
63                          state = 7;
64                      } else {
65                          result = true;
66                          state = 9;
67                      }
68                      break;
69
70                  case 7:     // Поиск очередного предмета
71                      if (T[i][sum] != T[i - 1][sum]) {
72                          positions.add(new Integer(i - 1));
73                          sum -= M[i - 1];
74                          state = 8;
75                      } else {
76                          state = 8;
77                      }
78                      break;
79
80                  case 8:
81                      i--;
82                      state = 6;
```

```
83                      break;
84                  }
85              }
```

Такая реализация автомата резко упрощает построение визуализатора, так как к каждому состоянию автомата (метке *case*) могут быть «привязаны» соответствующие иллюстрации и комментарии. В силу того, что реализация визуализатора выполняется с помощью указанного выше паттерна, «привязка» в данном случае осуществляется с помощью второго оператора *switch*. Таким образом, в визуализаторе используется два оператора *switch*, первый из которых реализует автомат, а второй применяется в формирователе иллюстраций и комментариев.

Отметим, что формализованный подход к построению логики визуализатора привел к тому, что отладка логики отсутствовала, а небольших изменений потребовала неформализуемая часть программы, связанная с построением иллюстраций и комментариев.

Аплет и исходный текст программы визуализатора приведены на сайте http://is.ifmo.ru/works/art_vis/.

2.2.3. Построение визуализаторов на основе автоматов Мура

Как уже отмечалось, в работе [58] было показано, что при программной реализации по схеме алгоритма может быть формально построены как автомат Мура, так и автомат Мили. Как отмечалось выше, в автоматах первого типа действия выполняются в вершинах, а в автоматах второго типа – на переходах. Для повышения наглядности и упрощения построения визуализатора в данном примере применяются автоматы Мура. При этом отметим, что число состояний в автоматах Мура обычно больше (не меньше), чем их число в эквивалентном автомате Мили.

8. Преобразование схемы алгоритма в граф переходов автомата Мура

В соответствии с методом, изложенным в работе [58], определим состояния, которые будет иметь автомат Мура, соответствующий этой схеме

алгоритма. Для этого присвоим номера последовательностям операторных вершин (последовательность может состоять и из одной вершины).

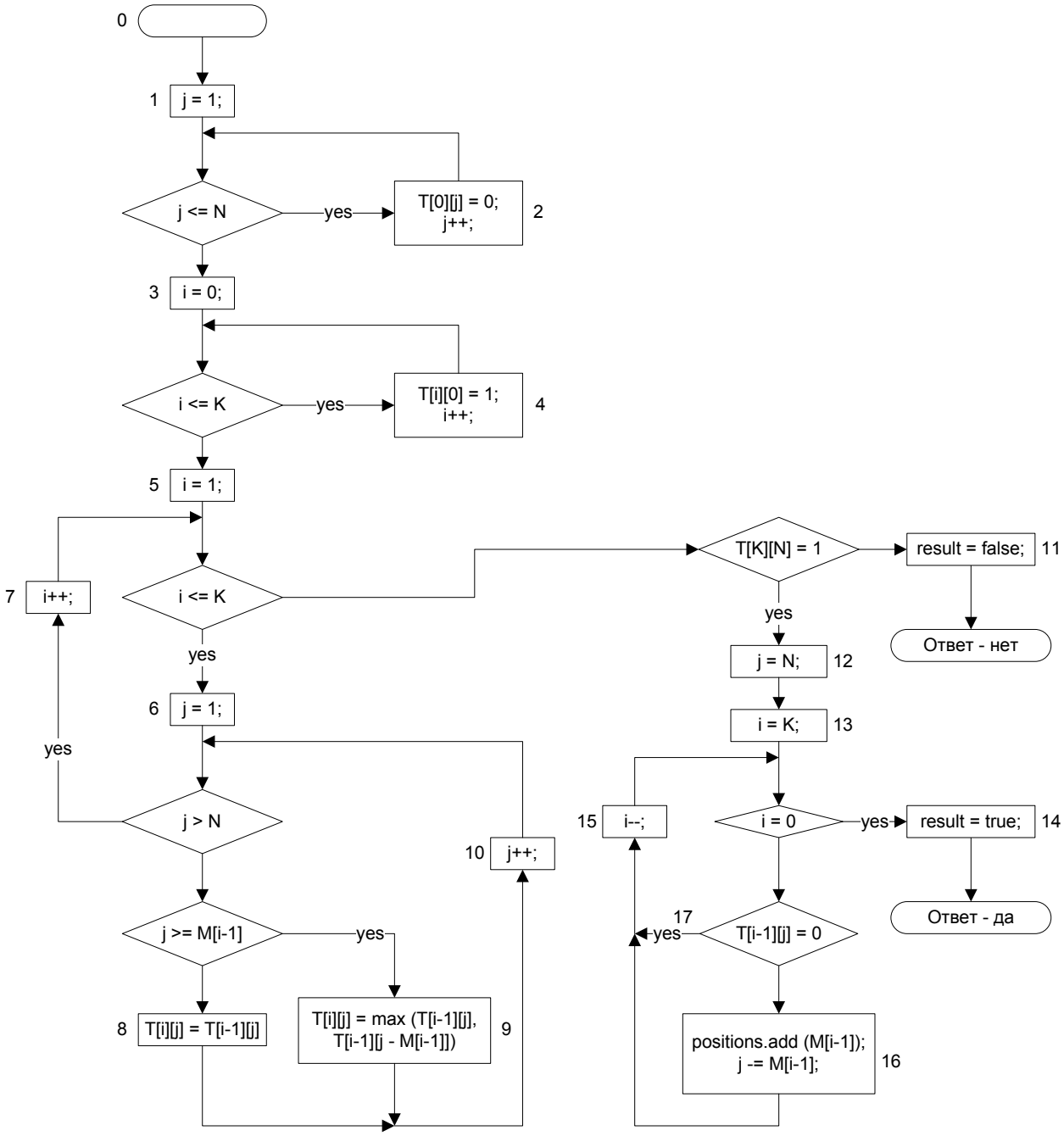

Рис. 20. Схема алгоритма решения задачи о рюкзаке

Исходя из изложенного метода, для схемы алгоритма на рис. 20 выделим шестнадцать состояний автомата с номерами 0 – 12, 14 – 16.

Число состояний, определенных описанным методом, бывает для целей визуализации недостаточным, так как в ряде случаев необходимо показывать

также условия и результаты некоторых действий. В рассматриваемом примере для этого введем состояния 13 и 17.

В рассматриваемом примере в состояниях 2 и 4 выполняется заполнение таблицы значений функции *T* начальными данными в первой строке и в первом столбце. В состояниях 6 – 10 реализуется алгоритм заполнения таблицы значений функции *T*. Состояние с номером 13 введено для визуализации перехода ко второму этапу алгоритма – поиску набора предметов. Состояния 16 и 17 отображают первую половину шага алгоритма обратного хода, причем более сложный переход соответствует включению предмета с номером *i* в результирующий набор, а второй переход – не включению предмета в этот набор. Состояние 15 соответствует второй половине шага второго этапа алгоритма. Таким образом, автомат визуализации будет иметь восемнадцать состояний, а его граф переходов, который строится за счет определения путей между выделенными точками на схеме алгоритма – восемнадцать вершин (Рис. 2).

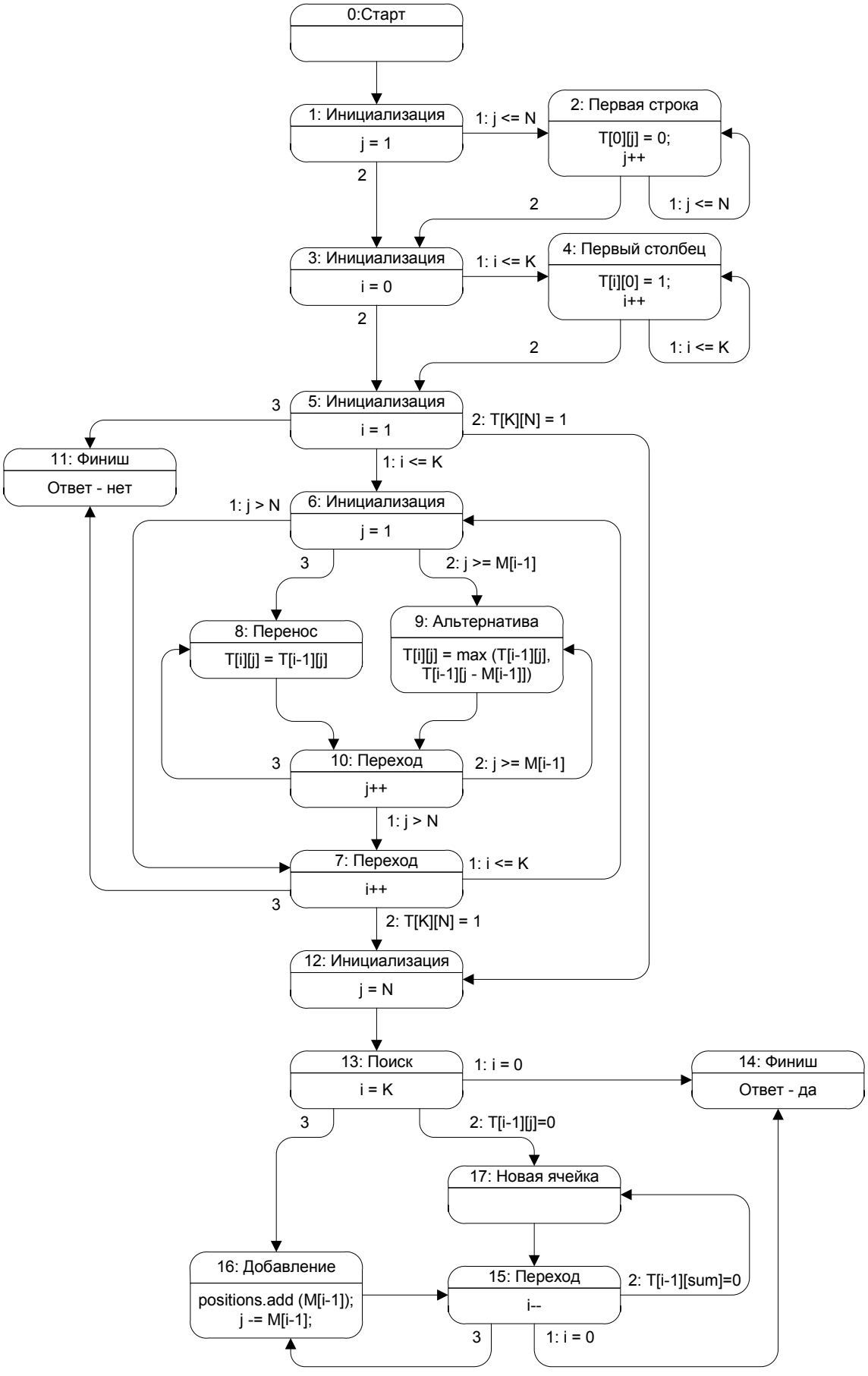

Рис. 21. Граф переходов для визуализации логики алгоритма

9. Формирование набора невизуализируемых состояний

Из анализа графа переходов (рис. 21) следует, что «неинтересными» (не изменяющими визуализацию) являются следующие состояния: 1, 3, 5, 6, 7, 10. Устранение таких состояний выполняется за счет того, что один шаг работы визуализатора состоит из нескольких переходов автомата, реализующего его логику.

Для исключения «неинтересных» состояний шаг визуализатора реализуется в программе следующим образом:

```
1    while состояние in (1, 3, 5, 6, 7, 10)
2      do сделать переход автомата
```

10.Выбор интерфейса визуализатора

Поскольку модификация метода не влияет на внешнее поведение визуализатора, то в настоящем примере предлагается использовать тот же интерфейс, что и на этапе 10 модификации метода построения визуализаторов на основе автоматов Мили (рис. 9).

11.Сопоставление иллюстраций и комментариев с состояниями автомата

Ввиду того, что при построении визуализатора используется автомат Мура, то будем в рассматриваемом состоянии отображать выполняемые в них действия.

При этом отметим, что серым цветом подсвечивается текущая ячейка таблицы (рис. 22, рис. 23), а черным – рабочие ячейки (рис. 24, рис. 25).

Состояние 0. Описано в разд. 2.2.2 (рис. 9).

Состояние 1 является «невизуализируемым», так как оно служебное.

Состояние 2. Заполнение нулями верхней строки таблицы *T* (рис. 22).

Рис. 22. Состояние 2

Состояние 3 является «невизуализируемым», так как оно служебное.

Состояние 4. Заполнение единицами левого столбца таблицы T (рис. 23).

Рис. 23. Состояние 4

Состояния 5 – 7 не визуализируются, так как являются служебными.

На рис. 24 приведена иллюстрация, соответствующая **состоянию 8**, в котором выполняется копирование значения из расположенной выше ячейки таблицы (помечена черным цветом) в ячейку, расположенную ниже (помечена серым цветом), поскольку рассматриваемый в данный момент предмет не может быть использован в наборе. Это имеет место в случае, когда сумма весов предметов для текущего столбца меньше веса предмета, соответствующего текущей строке.

На рис. 25 приведена иллюстрация к **состоянию 9**, на которой показано, что имеет место альтернатива в выборе значения, поскольку вес T (i, j) может быть получен двумя разными способами.

В первом случае текущий предмет включается в набор, а во втором рассматриваемый предмет в набор не включается. Наличие единицы в одной из двух черных (Рис. 7) ячеек соответствует одному из вариантов. Поскольку на Рис. 7 выполняется соотношение T (1, 4) = 1, то набор с весом четыре может быть получен с применением одного предмета. Поэтому набор с весом девять может быть получен из двух предметов за счет добавления второго предмета с весом пять. Следовательно, T (2, 9) = 1.

Рис. 24. Состояние 8

Рис. 25. Состояние 9

Состояние 10 является «невизуализируемым», так как оно служебное.

Состояние 11 соответствует случаю, когда решение задачи отсутствует. Это состояние является одним из конечных.

На этом первый этап алгоритма завершен.

Состояние 12 отображает проверку того, что искомый суммарный вес набора предметов может быть получен, поскольку значение в правой нижней ячейке равно единице (рис. 26). Это состояние является промежуточным между

прямым и обратным ходом алгоритма, на котором собственно и выполняется поиск результирующего набора.

Рис. 26. Состояние 12

Рассмотрим состояния автомата, соответствующие второму этапу алгоритма.

Состояния 13 – 17 автомата, образующие замкнутый контур на рис. 21, порождают искомый набор предметов. Опишем, как эти состояния иллюстрируются.

Состояния 13 и 15 отображают альтернативу первой половины шага при обратном ходе алгоритма (рис. 27) из серой ячейки в одну из черных. Вариант перехода в ячейку в том же столбце означает, что текущий предмет не включается в результирующий набор. Переход во вторую черную ячейку означает включение рассматриваемого предмета в набор. При этом отметим, что собственно переход всегда выполняется в ячейку, помеченную единицей. В случае если обе ячейки содержат по единице, то выбирается первый из описанных вариантов, поскольку выполняется поиск одного из решений, наличие же двух единиц означает существование альтернативного решения.

Состояния 16 и 17 отображают результат выбора на предыдущем шаге (рис. 28, рис. 29). При этом в **состоянии 16** предмет, соответствующий рассматриваемой строке, включается в набор, а в **состоянии 17** выполняется переход к следующей ячейке без включения предмета. Из левой части рисунков следует, что предмет с весом 7 включен в результирующий набор, в то время как предмет с весом 3 в результирующий набор не включен.

Рис. 27. Состояния 13,15

Рис. 28. Состояние 16

Рис. 29. Состояние 17

Состояние 14 отображает искомый результат – набор предметов, реализующих суммарный вес, равный 16 (рис. 30).

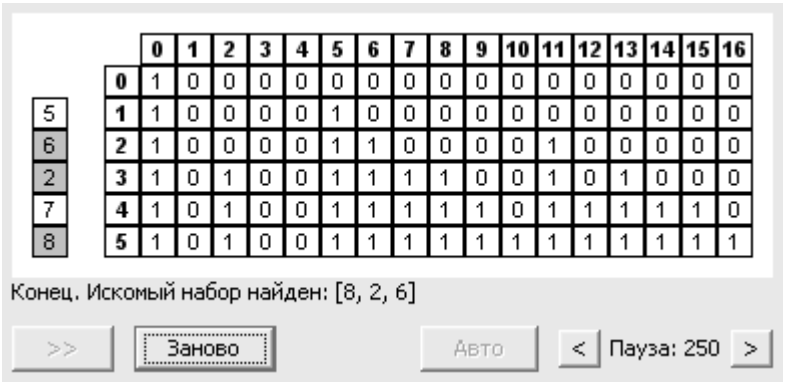

Рис. 30. Состояние 14

12.Выбор архитектуры программы визуализатора

Аналогично модификации метода построения визуализаторов на основе автоматов Мили, визуализатор предлагается реализовывать на основе паттерна *Модель – Вид – Контроллер* [52].

13.Программная реализация визуализатора

В работе [59] было предложено переходить формально и изоморфно от графа переходов автомата к его программной реализации на основе операторов *switch*. Для рассматриваемого примера результатом преобразования графа переходов (рис. 21) является текст программы, приведенный ниже:

```
1       private void makeAutomationStep() {
2           switch (g.state) {
3               case 0: // Начальное состояние
4                   goToAndPerformAction(1);
5                   break;
6
7               case 1:
8                   if (g.j <= g.N) {
9                       goToAndPerformAction(2);
10                  } else {
11                      goToAndPerformAction(3);
12                  }
13                  break;
14
15              case 2: // Заполнение 0-й строки
16                  if (g.j <= g.N) {
17                      goToAndPerformAction(2);
18                  } else {
19                      goToAndPerformAction(3);
20                  }
21                  break;
22
23              case 3:
24                  if (g.i <= g.K) {
25                      goToAndPerformAction(4);
26                  } else {
27                      goToAndPerformAction(5);
28                  }
```

```
29              break;
30
31          case 4: // Заполнение 0-го столбца
32              if (g.i <= g.K) {
33                  goToAndPerformAction(4);
34              } else {
35                  goToAndPerformAction(5);
36              }
37              break;
38
39          case 5:
40              if (g.i <= g.K) {
41                  goToAndPerformAction(6);
42              } else if (g.T[g.K][g.N] == 1) {
43                  goToAndPerformAction(12);
44              } else {
45                  goToAndPerformAction(11);
46              }
47              break;
48
49          case 6: // Переход к следующей строке
50              if (g.j > g.N) {
51                  goToAndPerformAction(7);
52              } else if (g.j >= g.M[g.i - 1]) {
53                  goToAndPerformAction(9);
54              } else {
55                  goToAndPerformAction(8);
56              }
57              break;
58
59          case 7:
60              if (g.i <= g.K) {
61                  goToAndPerformAction(6);
62              } else if (g.T[g.K][g.N] == 1) {
63                  goToAndPerformAction(12);
64              } else {
65                  goToAndPerformAction(11);
66              }
67              break;
68
69          case 8: // Перенос
70              goToAndPerformAction(10);
71              break;
72
73          case 9: // Альтернатива
74              goToAndPerformAction(10);
75              break;
76
77          case 10:// Переход к следующей строке
78              if (g.j > g.N) {
79                  goToAndPerformAction(7);
80              } else if (g.j >= g.M[g.i - 1]) {
81                  goToAndPerformAction(9);
82              } else {
83                  goToAndPerformAction(8);
84              }
85              break;
86
87          case 12:
88              goToAndPerformAction(13);
89              break;
90
91          case 13:
92              if (g.i == 0) {
93                  goToAndPerformAction(14);
```

```
94                } else if (g.T[g.i][g.sum] == g.T[g.i-1][g.sum]) {
95                    goToAndPerformAction(17);
96                } else {
97                    goToAndPerformAction(16);
98                }
99                break;
100
101           case 15:
102               if (g.i == 0) {
103                   goToAndPerformAction(14);
104               } else if (g.T[g.i][g.sum] == g.T[g.i-1][g.sum]) {
105                   goToAndPerformAction(17);
106               } else {
107                   goToAndPerformAction(16);
108               }
109               break;
110
111           case 16: // элемент найден
112               goToAndPerformAction(15);
113               break;
114
115           case 17:
116               goToAndPerformAction(15);
117               break;
118       }
119   }
120   private void goToAndPerformAction(int newstate) {
121       g.state = newstate;
122       switch (newstate) {
123           case 0: // Начальное состояние
124               g.state = 1;
125               break;
126
127           case 1:
128               g.j = 1;
129               break;
130
131           case 2: // Заполнение 0-й строки
132               g.T[0][g.j] = 0;
133               g.j++;
134               break;
135
136           case 3:
137               g.i = 0;
138               break;
139
140           case 4: // Заполнение 0-го столбца
141               g.T[g.i][0] = 1;
142               g.i++;
143               break;
144
145           case 5:
146               g.i = 1;
147               break;
148
149           case 6: // Переход к следующей строке
150               g.j = 1;
151               break;
152
153           case 7:
154               g.i++;
155               break;
156
157           case 8: // Перенос
158               g.T[g.i][g.j] = g.T[g.i - 1][g.j];
```

```
159                 break;
160
161             case 9: // Альтернатива
162                 g.T[g.i][g.j] = Math.max(g.T[g.i - 1][g.j],
163                                          g.T[g.i - 1][g.j - g.M[g.i - 1]]);
164                 break;
165
166             case 10:// Переход к следующей строке
167                 g.j++;
168                 break;
169
170             case 12:
171                 g.sum = g.N;
172                 break;
173
174             case 13:
175                 g.i = g.K;
176                 break;
177
178             case 15:
179                 g.i--;
180                 break;
181
182             case 16: // элемент найден
183                 g.positions.add(new Integer(g.i-1));
184                 g.sum -= g.M[g.i-1];
185                 break;
186         }
187     }
```

В приведенной реализации используется два оператора *switch*. Первый оператор реализует ребра графа переходов, а второй – действия в вершинах.

Такая реализация автомата резко упрощает построение визуализатора, так как к каждому состоянию автомата (метке *case*) могут быть «привязаны» соответствующие иллюстрации и комментарии. В силу того, что реализация визуализатора выполняется с помощью указанного выше паттерна, «привязка» в данном случае осуществляется с помощью второго оператора *switch*. Таким образом, в визуализаторе используется **три** оператора *switch*, первый из которых реализует переход автомата, второй реализует действия в вершинах, а третий применяется в формирователе иллюстраций и комментариев.

Полный исходный текст программы визуализатора также приведен по адресу http://is.ifmo.ru/works/art_vis/.

2.2.4. Особенности разновидностей построения визуализаторов

В завершение описания формализованного метода построения визуализаторов приведем сравнительную таблицу двух модификаций метода (табл. 4).

Таблица 4. Сравнение модификаций метода

Особенность	**Автоматы Мили**	**Автоматы Мура**
Число состояний автомата	10	18
Исключение иллюстраций	«Неинтересные» переходы	«Неинтересные» состояния
Иллюстрации	Каждому состоянию соответствует одна или несколько иллюстраций	Каждому состоянию соответствует одна иллюстрация
Размер кода автомата (строк)	85	187

Рассмотрим различия между модификациями.

- Как и было отмечено в разд. 2.2.2, автомат Мили содержит меньшее число состояний. Таким образом, визуализаторы, построенные на основе автоматов Мили, всегда будут иметь меньшее число строк для реализации автомата.
- Несмотря на то, что автоматы Мура имеют больше состояний, наличие единственной иллюстрации в каждом «интересном» состоянии упрощает построение формирователя иллюстраций, поскольку формирователь иллюстраций реализовывается посредством единственного оператора *switch*. В случае же автоматов Мили, некоторых состояниях приходится формировать несколько иллюстраций (в соответствии с количеством переходов в иллюстрируемое состояние), что усложняет логику формирователя иллюстраций.

Несмотря на различные реализации, число иллюстраций и внешнее поведение визуализаторов совпадают, что предоставляет свободу разработчику визуализатора в выборе модификации используемого метода.

После формулировки методов формализованного построения визуализаторов проведем сравнительный анализ формализованного метода на основе конечных автоматов, эвристического автоматного метода и полностью эвристического метода, рассмотренного в первой главе.

2.3. Сравнение традиционного и автоматных (эвристического и формализованного) методов разработки визуализаторов алгоритмов

В настоящем разделе на примере простого алгоритма «пузырьковая сортировка» выполнено сравнение трех методов построения логики визуализаторов. При этом рассматриваются следующие методы:

- эвристический (без использования автоматов для описания поведения визуализатора);
- непосредственное построение автомата по словесному описанию алгоритма;
- формализованное построение автомата по схеме алгоритма.

В заключение раздела приведены рекомендации по их использованию.

2.3.1. Алгоритм «пузырьковая сортировка»

Задан массив a из n элементов, который требуется отсортировать по возрастанию. Слева направо поочередно сравниваются пары соседних элементов (a_1 и a_2), и если элемент a_1 больше a_2, то они меняются местами. После этого выбирается следующая пара элементов (a_2 и a_3), и она упорядочивается, как указано выше. Аналогичные действия выполняются для всех оставшихся пар. После первого прохода на последней позиции массива (n-ой позиции) будет стоять максимальный элемент («Всплыл первый пузырек»). Поскольку максимальный элемент уже размещен, то во втором проходе выполняется сортировка (n – 1)-го элемента. После этого сортируются n – 2 элементов и т. д. Таким образом, за n – 1 проход массив будет отсортирован.

2.3.2. Интерфейс визуализатора

На рис. 31 приведен интерфейс визуализатора, который используется для всех трех рассматриваемых ниже методов.

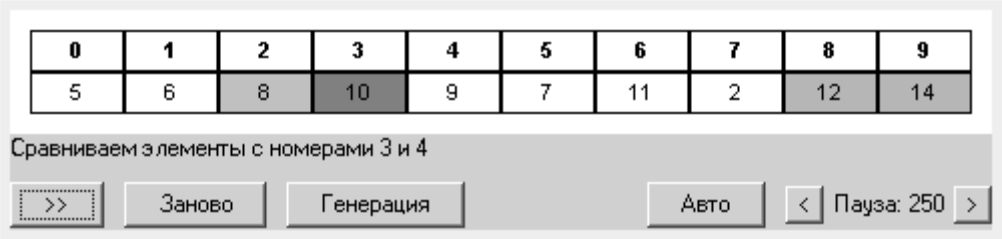

Рис. 31. Интерфейс визуализатора

2.3.3. Эвристический подход

Метод широко используется на практике и состоит в эвристическом построении по словесному алгоритму реализующей его программы с дальнейшим, также эвристическим ее преобразованием в визуализатор. При этом описание логики визуализаторов выполняется без применения конечных автоматов [78].

Программа, реализующая заданный алгоритм традиционным путем, приведена в листинге 2.

Листинг 2. Программа, реализующая алгоритм «пузырьковая сортировка» традиционным путем

```
1   public class Bubble {
2
3       private static int a[] = {4, 3, 7, 1, 9, -2};
4       private static int n = a.length;
5       private static int i = 0;
6       private static int j = 0;
7
8       public static void main(String[] args) {
9           for (int i = 1; i < n; i++) {
10              for (int j = 1; j < n - i + 1; j++) {
11                  if (a[j-1] > a [j]) {
12                      out();                          // Пошаговый вывод
13                      swap(j,j-1);
14                  }
15              }
16          }
17          out();
18      }
19
20      // Обмен и вывод
21
22      private static void swap (int i, int j) {
23          int t = a[i];
24          a[i] = a[j];
25          a[j] = t;
26      }
27
28      private static void out () {
29          for (int i = 0; i < n; i++) {
30              System.out.print("" + a[i] +" ");
31          }
```

```
32          System.out.println();
33      }
34  }
```

Правильность работы этой программы подтверждается протоколом, выводящим последовательные состояния массива при перестановках элементов:

```
4 3 7 1 9 -2
3 4 7 1 9 -2
3 4 1 7 9 -2
3 4 1 7 -2 9
3 1 4 7 -2 9
3 1 4 -2 7 9
1 3 4 -2 7 9
1 3 -2 4 7 9
1 -2 3 4 7 9
-2 1 3 4 7 9
```

Отметим, что такой же протокол будут иметь программы, построенные на основе других методов.

Для построения визуализатора в целом эту программу необходимо эвристически связать с пользовательским интерфейсом.

2.3.4. Автоматный подход

Суть этого метода состоит в том, что по словесной формулировке алгоритма строится непосредственно граф переходов автомата, задающего логику визуализатора. Привязка этой программы к интерфейсу осуществляется значительно проще, чем в предыдущем случае, так как в модели, описывающей логику визуализатора, присутствуют состояния, к которым «весьма легко» привязать визуализируемую информацию.

На рис. 32 приведен граф переходов автомата, построенного непосредственно по словесной формулировке алгоритма.

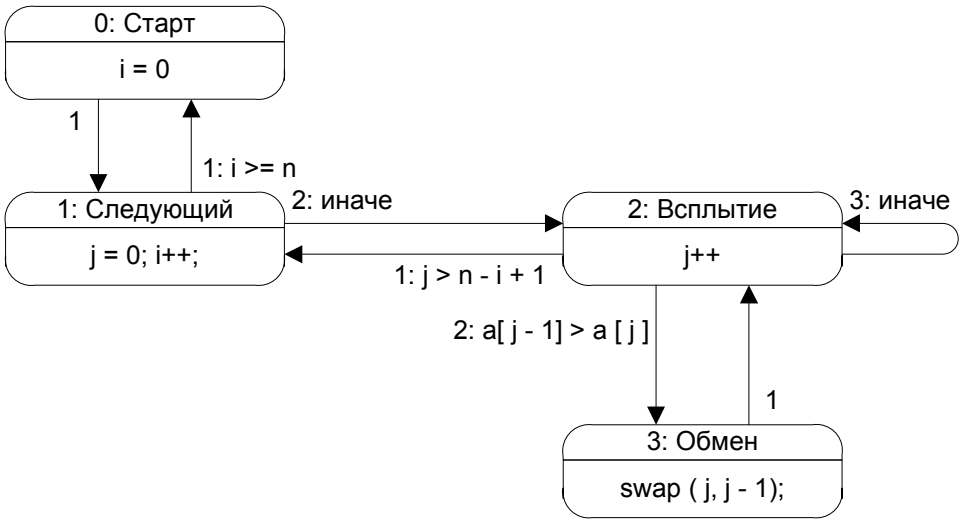

Рис. 32. Граф переходов, построенный непосредственно по словесному описанию алгоритма

Программа, изоморфная этому графу переходов, приведена в листинге 3.

Листинг 3. Программа, реализующая алгоритм «пузырьковая сортировка», построенная по графу переходов

```
1   public class BubbleA {
2
3       public static final int STATE_INIT   = 0;
4       public static final int STATE_NEXT   = 1;
5       public static final int STATE_EMERGE = 2;
6       public static final int STATE_SWAP   = 3;
7
8       private static int state = STATE_INIT;
9       private static int a[] = {4, 3, 7, 1, 9, -2};
10      private static int n = a.length;
11      private static int i = 0;
12      private static int j = 0;
13
14      public static void main(String[] args) {
15          do {
16              nextStep();
17          } while (state != STATE_INIT);
18          out();
19      }
20
21      private static void nextStep() {
22          switch(state) {
23              case STATE_INIT:
24                  i = 0;
25                  state = STATE_NEXT;
26                  break;
27
28              case STATE_NEXT:
29                  j = 0;
30                  i++;
31
32                  if (i >= n) {
33                      state = STATE_INIT;
34                  } else {
35                      state = STATE_EMERGE;
```

```
36                  }
37                  break;
38
39              case STATE_EMERGE:
40                  j++;
41
42                  if (j >= n - i + 1) {
43                      state = STATE_NEXT;
44                  } else if (a[j-1] > a [j]) {
45                      state = STATE_SWAP;
46                  }
47                  break;
48
49              case STATE_SWAP:
50                  // Пошаговый вывод
51                  out();
52                  swap(j,j-1);
53                  state = STATE_EMERGE;
54                  break;
55          }
56      }
57
58          // Обмен и вывод
59          ...
60  }
```

Построение визуализатора по этой программе выполняется на основе подхода, как изложено в работе [70].

2.3.5. Формализованное построение автомата по схеме алгоритма

Этот метод состоит в следующем:

- по словесному описанию алгоритма эвристически строится реализующая его программа;
- по программе эвристически строится схема алгоритма;
- по схеме алгоритма формально строится граф переходов конечного автомата;
- граф переходов формально и изоморфно реализуется соответствующей программой;
- в построенную программу, содержащую «состояния», вводятся компоненты, обеспечивающие визуализацию алгоритма.

Программа для рассматриваемого алгоритма приведена в листинге 2. На рис. 33 приведена схема алгоритма, построенная по этой программе.

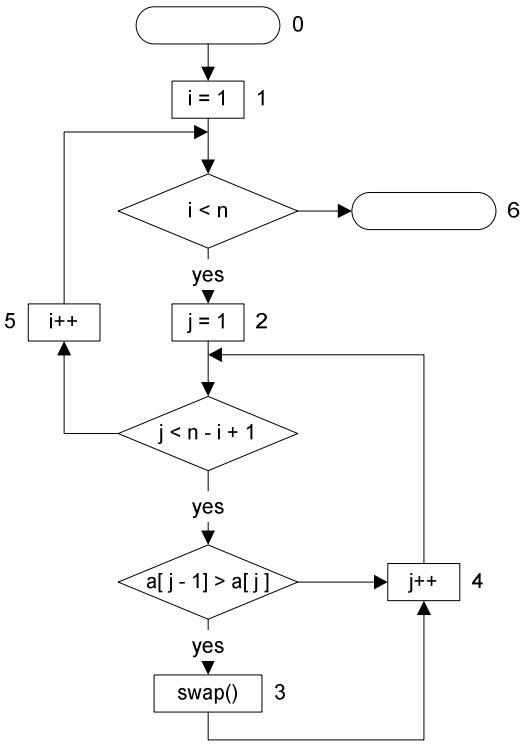

Рис. 33. Схема алгоритма «пузырьковая сортировка»

В работе [58] описан метод преобразования схемы алгоритма в граф переходов (в данном случае автомата Мура). На рис. 34 приведен этот граф, построенный на основе указанного метода.

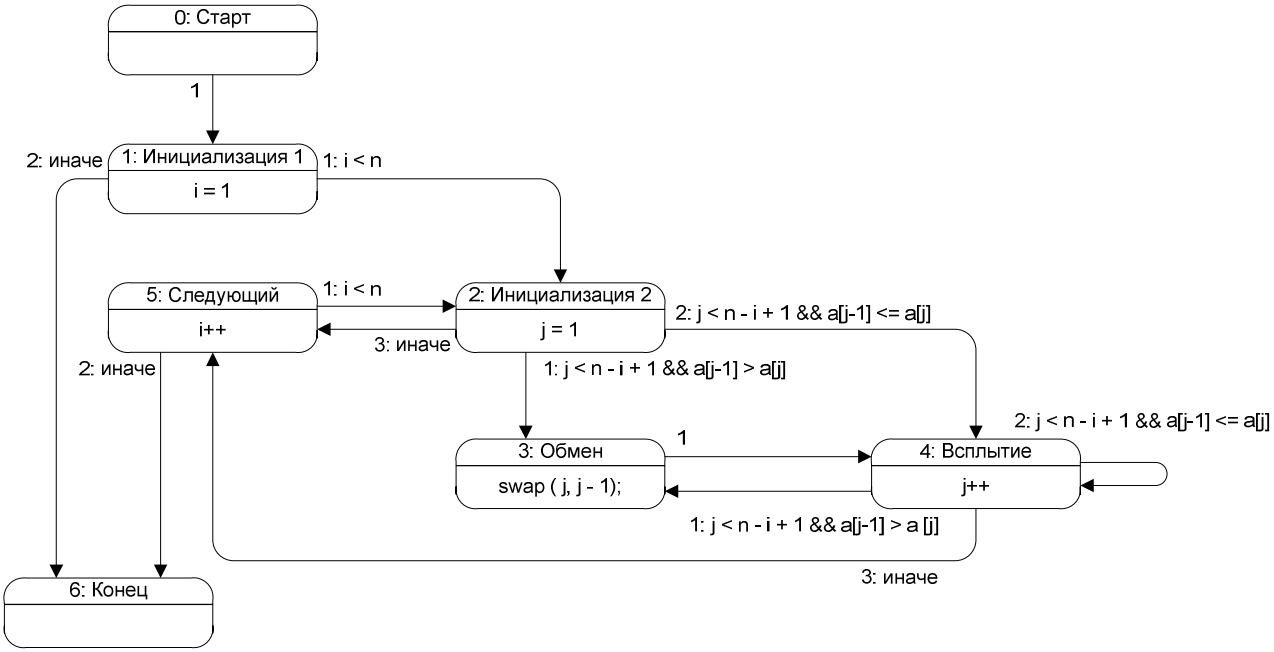

Рис. 34. Автомат Мура, построенный по схеме алгоритма «пузырьковая сортировка»

Программа, изоморфная этому графу переходов, приведена в листинге 4.

Листинг 4. Программа, изоморфная графу переходов, построенному формально по схеме алгоритма

```
1   public class BubbleAFormal {
2
3       public static final int STATE_START  = 0;
4       public static final int STATE_INIT1  = 1;
5       public static final int STATE_INIT2  = 2;
6       public static final int STATE_SWAP   = 3;
7       public static final int STATE_EMERGE = 4;
8       public static final int STATE_NEXT   = 5;
9
10      private static int state = STATE_START;
11      private static int a[] = {4, 3, 7, 1, 9, -2};
12      private static int n = a.length;
13      private static int i = 0;
14      private static int j = 0;
15
16      public static void main(String[] args) {
17          do {
18              nextStep();
19          } while (state != STATE_START);
20          out();
21      }
22
23      private static void nextStep() {
24          switch(state) {
25              case STATE_START:
26                  state = STATE_INIT1;
27                  break;
28
29              case STATE_INIT1:
30                  i = 1;
31                  if (i < n) {
32                      state = STATE_INIT2;
33                  } else {
34                      state = STATE_START;
35                  }
36                  break;
37
38              case STATE_INIT2:
39                  j = 1;
40                  if (j < n - i + 1 && a[j-1] > a [j]) {
41                      state = STATE_SWAP;
42                  } else if (j < n - i + 1 && a[j-1] <= a [j]) {
43                      state = STATE_EMERGE;
44                  } else {
45                      state = STATE_NEXT;
46                  }
47                  break;
48
49              case STATE_SWAP:
50                  out();                          // Пошаговый вывод
51                  swap(j,j-1);
52                  state = STATE_EMERGE;
53                  break;
54
55              case STATE_EMERGE:
56                  j++;
57
58                  if (j < n - i + 1 && a[j-1] > a [j]) {
59                      state = STATE_SWAP;
60                  } else if (j < n - i + 1 && a[j-1] <= a [j]) {
61                      state = STATE_EMERGE;
```

```
62                  } else {
63                      state = STATE_NEXT;
64                  }
65                  break;
66
67              case STATE_NEXT:
68                  i++;
69
70                  if (i < n) {
71                      state = STATE_INIT2;
72                  } else {
73                      state = STATE_START;
74                  }
75                  break;
76
77          }
78      }
79
80      private static void swap (int i, int j) {
81          int t = a[i];
82          a[i] = a[j];
83          a[j] = t;
84      }
85
86      private static void out () {
87          for (int i = 0; i < n; i++) {
88              System.out.print("" + a[i] +" ");
89          }
90          System.out.println();
91      }
92  }
```

2.3.6. Сравнение методов

В приведенном примере рассмотрены три подхода к построению визуализатора. В первом подходе визуализация работы алгоритма достигается за счет протоколирования. Такой вид визуализации не удовлетворяет основным требованиям к визуализатору, изложенным в разд. 1.1.

Эвристическое построение автомата, реализующего алгоритм привело к построению визуализатора, однако процесс построения такого автомата не является формализованным, следовательно такой подход не может считаться методом.

Формализованный метод также привел к построению визуализатора, однако число состояний автомата при таком подходе оказалось большим.

На основе изложенного можно сделать следующие выводы:

- несмотря на большую распространенность «эвристический подход к построению визуализаторов алгоритмов» методом не является;

- для алгоритмов дискретной математики явное выделение состояний может быть выполнено сравнительно редко. Однако когда это удается (как в рассмотренном примере), применение автоматного метода для построения визуализаторов весьма эффективно, так как если состояния выделяются естественным образом, то это приводит к сравнительно небольшой сложности графа переходов;
- метод построения визуализаторов на основе формализованного построения графа переходов по схеме алгоритма совмещает достоинства предыдущих двух методов: по алгоритму эвристически строится программа, что характерно для традиционного программирования, а граф переходов автомата получается по этой программе формально. Универсальность формализованного метода построения логики визуализаторов приводит к усложнению графа переходов и программы, построенной на его основе.

Дальнейшее совершенствование методов построения визуализаторов связано, как с автоматизацией третьего метода, так и с его развитием в части построения автоматов для визуализации алгоритма не только при его прохождении в прямом направлении, но и в обратном [74].

Выводы по главе 2

1. В главе рассмотрен эвристический автоматный подход к построению визуализаторов, приведено обоснование того, что этот подход методом не является.
2. Также был сформулирован формализованный метод построения визуализаторов алгоритмов на основе конечных автоматов. При этом рассмотрены две модификации метода, проведено их сравнение. Метод был подкреплен примерами, явно демонстрирующими работоспособность метода. Визуализаторы, построенные при помощи разных модификаций формализованного метода, совпали по

внешнему поведению, что показывает равнозначность двух предложенных модификаций.

3. Проведен сравнительный анализ эвристического и эвристического автоматного подходов и формализованного метода, который показал преимущество формализованного метода.
4. Было показано, что формализованное построение графа переходов по программе превращает подход к построению визуализаторов в метод.

ГЛАВА 3. МЕТОД ПОСТРОЕНИЯ ВИЗУАЛИЗУАЛИЗАТОРОВ НА ОСНОВЕ СИСТЕМЫ ВЗАИМОДЕЙСТВУЮЩИХ АВТОМАТОВ

В главе 2 был рассмотрен метод построения визуализаторов на основе автоматного подхода. При построении визуализатора предполагалось, что визуальное представление алгоритма состоит из последовательности этапов (шагов). Однако не всегда достаточно одного уровня детализации. При построении сложных алгоритмов часто возникает необходимость предоставлять обучаемому возможность пропуска деталей с целью акцентирования внимания на крупных шагах алгоритма. Особенно это важно когда малые шаги визуализатора многократно повторяются.

Далее будет рассмотрена модификация метода, представленного в разд. 2.2. Результатом модификации будет являться метод построения визуализаторов алгоритмов на основе систем взаимодействующих автоматов. Важной особенностью визуализаторов, построенных этим методом, будет возможность отображения хода алгоритма на разных уровнях визуализации. Обеспечение различных уровней визуализации производится за счет введения *крупных* шагов визуализатора. *Крупным шагом* визуализатора называется шаг, состоящий из набора элементарных шагов визуализатора.

В разд. 3.1 рассматривается метод построения визуализаторов алгоритмов дискретной математики на основе системы взаимодействующих автоматов. В разд. 3.2 действие метода демонстрируется на примере «задачи о рюкзаке». В разд. 3.3 производится сравнение с базовым методом.

3.1. Метод построения визуализатора

В основе метода построения визуализаторов на основе системы взаимодействующих автоматов лежит метод, описанный в разд. 2.2. В отличие от базового метода в подходе, рассматриваем в настоящей главе, схемы алгоритма преобразовывается не в один автомат, а в систему

взаимодействующих автоматов. При этом один из автоматов является главным, и именно этот автомат взаимодействует с пользовательским интерфейсом, в то время как остальные автоматы являются подчиненными и реализуют логику построения многоуровневой визуализации.

С целью обеспечения возможности реализации крупных шагов в последовательность действий, приведенной во второй главе, вводятся дополнительные шаги. Отметим, что здесь рассматривается расширение метода на основе автоматов Мура. Приведенная ниже модификация также может быть также применена и к автоматам Мили.

3.1.1. Изменения метода для обеспечения системы взаимодействующих автоматов

Важной отличительной особенностью визуализаторов, построенных на основе системы взаимодействующих автоматов, является тот факт, что каждому состоянию главного автомата могут соответствовать различные состояния автоматов, реализующие малые шаги и этапы алгоритма. Предположим, что при преобразовании схемы алгоритма в систему взаимодействующих автоматов было получено N автоматов Мура. Введем обозначения:

- A_i – автомат с номером i ($i = 1..N$);
- s_i – состояние автомата A_i ;
- $S = \{s_1..s_N\}$ – состояние системы взаимодействующих автоматов;
- V_j – интересное (визиализируемое) состояние системы с номером j ($j = 1..K$);
- F_S (S) – функция перевода состояния системы автоматов в статическую иллюстрацию.

Для реализации визуализатора в метод, приведенный во второй главе, вносятся следующие изменения:

a) **После построения программы по схеме алгоритма на шестом этапе проводится преобразование программы с целью выделения крупных шагов в процедуры**.

b) При построении схемы алгоритма по программе на седьмом этапе строится не одна, а несколько схем: одна схема строится для главного алгоритма и по одной схеме для каждой процедуры.

c) На восьмом этапе проводятся действия для каждого автомата.

d) На девятом этапе при формировании набора неинтересных состояний происходит формирование набора неинтересных состояний системы автоматов $\{S_j\}_{j=1}^{K}$.

e) На одиннадцатом этапе при формировании иллюстраций они сопоставляются не состояниям автомата, а состояниям системы автоматов.

3.1.2. Формализация взаимодействия автоматов

Как отмечено выше, схема алгоритма после разбиения на процедуры преобразуется в систему взаимодействующих автоматов Мура. При этом встает задача преобразования вызовов вложенных процедур в механизм взаимодействие между автоматами. Для этой задачи предлагается следующее решение. Каждый вызов вложенной процедуры преобразовывается в запуск вложенного автомата. При этом используется следующее преобразование (рис. 35).

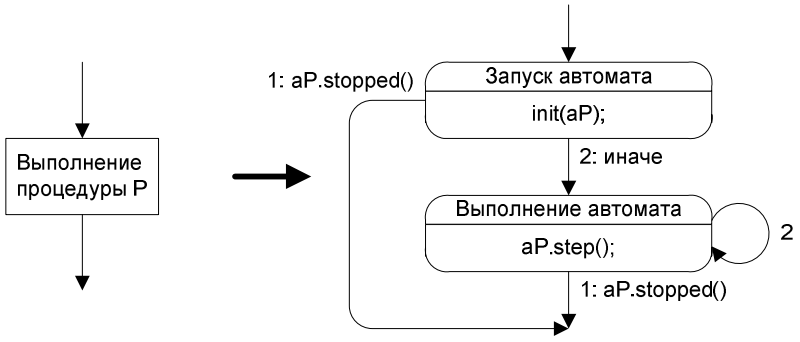

Рис. 35. Преобразование вызова вложенной процедуры

На этом рисунке используются следующие обозначения:

- *P* – процедура, выполняющая мелкие шаги алгоритма;
- *aP* – автомат, полученный преобразованием схемы алгоритма процедуры *P* в граф переходов автомата Мура;
- *aP.stopped()* – булево выражение, которое принимает значение истины в случае окончания работы автомата;
- *aP.step()* – выполнение одного перехода вложенного автомата Мура.

3.1.3. Введение уровней визуализации

В дополнение к стандартным преобразованиям в метод добавляется дополнительное действие, необходимое для введения в интерфейс визуализатора возможности пропуска малых шагов в рамках крупных:

f) Каждому из полученных автоматов Мура сопоставляется его уровень. При этом чем больше уровень автомата, тем более крупные шаги алгоритма он реализует.

Уровни автомата позволяют реализовать возможность для учащегося пропускать некоторые шаги алгоритма для более быстрого перехода к интересующим шагам, и тем самым не тратить время на малые, более понятные шаги. Для примера приведем алгоритм, состоящий из трех этапов, в котором второй этап разделен на более мелкие шаги (рис. 36).

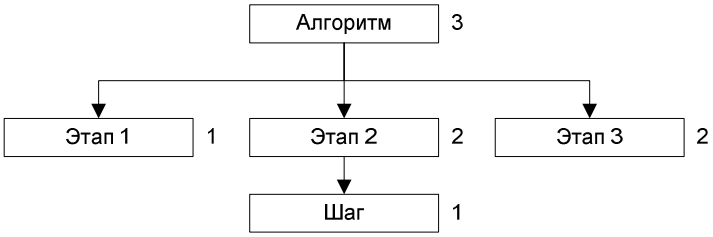

Рис. 36. Назначение уровней каждому автомату

В этом примере (рис. 36) визуализатор алгоритма будет демонстрировать следующее поведение.

- Этап 1 – шаги этой части алгоритма являются детальными и необходимы лишь в ситуации подробного изучения алгоритма.
- Этапы 2 и 3 являются менее детальными и рассматриваются на более высоком уровне изучения алгоритма. При этом этап 2 состоит из

шагов, каждый из которых, в свою очередь, является достаточно детальным. Поэтому автомат, реализующий шаги этапа 2, имеет первый уровень.

Для реализации многоуровневой системы визуализации используется следующий подход. При желании учащегося пропустить какие-то части алгоритма он выбирает уровень детализации, который он хочет пропустить одной из кнопок (рис. 37) и автомат визуализатора совершает переходы до тех пор, пока не будет достигнуто следующее состояние автомата требуемого уровня. На псевдокоде [2] этот алгоритм записывается следующим образом.

```
1  if уровень автомата >= уровень пропуска
2    then текущее состояние ← состояние автомата
3         while состояние автомата = текущее состояние
4           do выполнить переход автомата
```

Рис. 37. Кнопки управления уровнем шага

При этом наличие кнопок пропуска деталей определенного уровня и их число не является строгим и зависит от конкретного визуализатора. Так в примере, рассматриваемом далее в этой главе, визуализатор имеет три уровня детализации, в то время как другие визуализаторы могут иметь другое число уровней.

3.1.4. Внесение изменений в формирователь иллюстраций

В терминах приведенных выше обозначений формирователь иллюстраций должен реализовывать функцию $F_S(S) = F_S(s_1, s_2, \ldots, s_N)$. В базовом методе, приведенном в главе 2, формирователь иллюстраций является частным случаем формирователя, используемого в рассматриваемом методе. В то время как в методе, изложенном в главе 2, формирователь иллюстраций реализовывается посредством единственного оператора *switch*, в настоящем методе реализация формирования статических иллюстраций достигается посредством системы вложенных операторов *switch*.

В общем случае формирователь иллюстраций должен строить статическую иллюстрацию для каждого состояния системы автоматов. Однако, поскольку до построения формирователя иллюстраций строится список интересных состояний $\{V_j\}_{j=1}^{K}=\{\{v_1,v_2,\ldots,v_N\}_j\}_{j=1}^{K}$, то при формировании операторов *switch* необходимо построить иллюстрации лишь для интересных состояний. Таким образом, формирователь иллюстраций реализуется следующим образом (листинг 5).

Листинг 5. Формирование иллюстраций в системе взаимодействующих автоматов

```
1    switch s1
2    case v1_1:
3    switch s2:
4      case v2_1:
5        ...
6        switch sn:
7        case vn_1:
8          сформировать иллюстрацию F(v1_1,...,vn_1)
9          ...
```

Несмотря на увеличение кода формирователя иллюстраций, число иллюстраций может остаться таким же, как и визуализатора, построенного традиционным методом, изложенным во второй главе. Исключение составляют случаи, когда при построении многоуровневой визуализации требуется ввести дополнительные иллюстрации, обеспечивающие специальные пояснения начала или конца определенных этапов алгоритма.

3.1.5. Введение новых этапов в базовый метод

После приведенных изменений этапы базового метода преобразуются следующим образом:

1. **Постановка задачи**.
2. **Решение задачи** (в словесно-математической форме).
3. **Выбор визуализируемых переменных**.

4. **Анализ алгоритма для визуализации**. Анализируется решение с целью определения того, что и как отображать на экране.
5. **Реализация алгоритма решения задачи**.
6. **Реализация алгоритма на выбранном языке программирования**. На этом шаге производится реализация алгоритма, его отладка и проверка работоспособности.
7. **Преобразование программы с целью выделения крупных шагов в процедуры.**
8. **Построение схем алгоритма по программе: одна схема для главного алгоритма и по одной схеме для каждой вложенной процедуры**.
9. **Преобразование схем алгоритма в графы переходов системы автоматов Мура**.
10. **Формирование набора невизуализируемых состояний для каждого автомата**.
11. **Назначение уровня детализации каждому из автоматов.**
12. **Выбор интерфейса визуализатора**.
13. **Сопоставление иллюстраций и комментариев с состояниями автоматов**.
14. **Выбор архитектуры программы визуализатора**.
15. **Программная реализация визуализатора**.

Визуализатор, построенный указанным методом, реализует возможность перехода по крупным шагам алгоритма. При этом у учащегося появляется возможность просмотра визуализации на различных уровнях детализации.

3.2. Построение визуализатора алгоритма «задача о рюкзаке» с использованием взаимодействующих автоматов

Продемонстрируем построение визуализатора алгоритма на основе системы взаимодействующих автоматов на примере рассмотренной выше «задачи о рюкзаке».

7. Преобразование программы с целью выделения крупных шагов в процедуры

Как отмечено выше, шаги с первого по шестой не изменяются, поэтому начнем с программы, реализующей алгоритм (листинг 1). После выделения основных этапов алгоритма в процедуры получаем следующий код на языке *Java* (листинг 6).

Листинг 6. Преобразованная программа с выделенными процедурами

```
1  /**
2   * @param N суммарный вес предметов в рюкзаке
3   * @param M массив весов предметов
4   * @return список номеров предметов, сумма весов которых равна N,
5   *         либо null, если это невозможно.
6   */
7  private static Integer[] solve(int N, int[] M) {
8    // Число предметов
9    int K = M.length;
10   // Массив T
11   int[][] T = new int[K + 1][N + 1];
12   // Результирующие номера
13   Collection<Integer> positions = new ArrayList<Integer>();
14   // Результат работы true, если суммарный вес можно получить
15   boolean result;
16
17   // Определение начальных значений функции T
18   firstRow(T);
19   firstColumn(T);
20
21   // Построение таблицы значений функции T
22   fillTable(M, T);
23
24   // Определение набора предметов
25   if (T[K][N] == 1) {
26     // Решение существует
27     searchResult(N, M, T, positions);
28     // Решение найдено
29     result = true;
30   } else {
31     result = false;
32   }
33
34   // Если решение найдено, то оно возвращается, иначе возвращается null
```

```
35    return (Integer[])(result ? positions.toArray(new Integer[0]) : null);
36  }
37
38  private static void firstRow(int[][] T) {
39    for (int j = 1; j < T[0].length; j++) {
40      T[0][j] = 0;
41    }
42  }
43
44  private static void firstColumn(int[][] T) {
45    for (int i = 0; i < T.length; i++) {
46      T[i][0] = 1;
47    }
48  }
49
50  private static void fillTable(int[] M, int[][] T) {
51    for (int i = 1; i < T.length; i++) {
52      for (int j = 1; j < T[i].length; j++) {
53        if (j >= M[i - 1]) {
54          T[i][j] = Math.max(T[i - 1][j], T[i - 1][j - M[i - 1]]);
55        } else {
56          T[i][j] = T[i - 1][j];
57        }
58      }
59    }
60  }
61
62  private static void searchResult(int N, int[] M, int[][] T,
63                                   Collection<Integer> positions) {
64    int sum = N;
65    for (int i = T.length - 1; i >= 1; i--) {
66      if (T[i][sum] != T[i - 1][sum]) {
67        positions.add(i);
68        sum -= M[i - 1];
69      }
70    }
71  }
```

8. Построение схем алгоритма по программе

Схемы алгоритма, соответствующие тексту программы приведены на рис. 38 – 43.

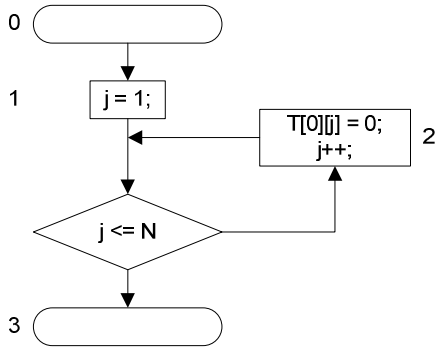

Рис. 38. Схема заполнения первой строки

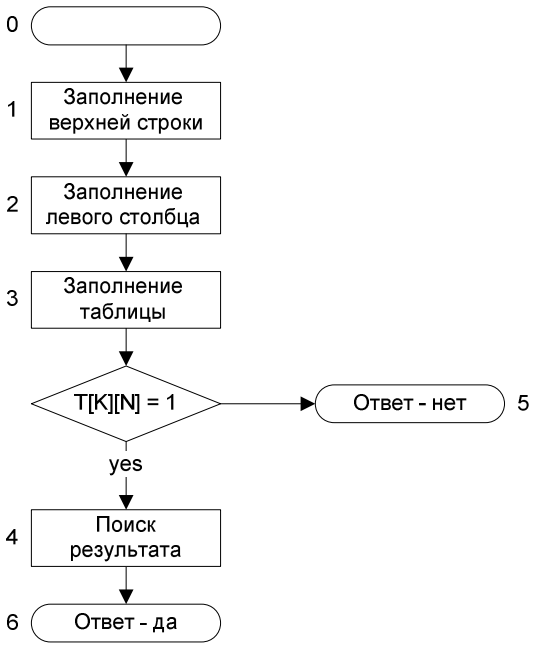

Рис. 39. Схема главного алгоритма

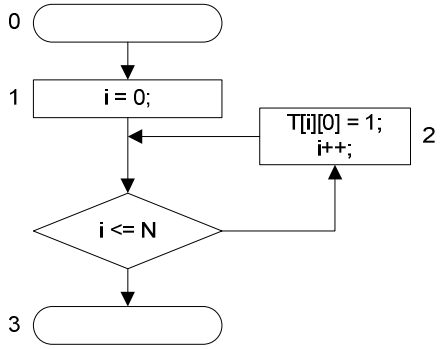

Рис. 40. Схема заполнения первого столбца

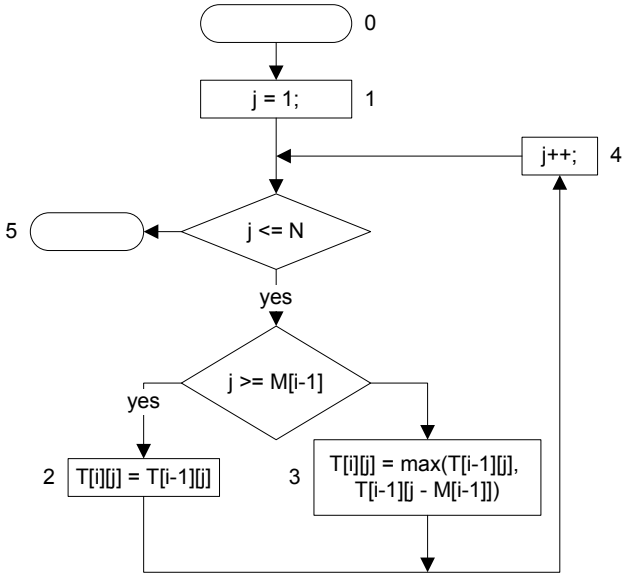

Рис. 41. Схема алгоритма заполнения строки *i*

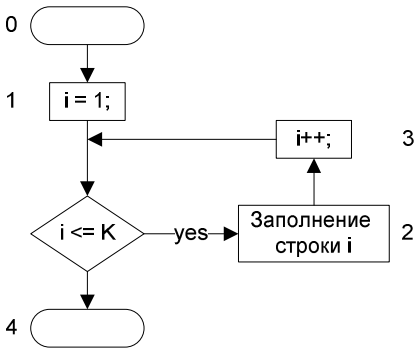

Рис. 42. Схема алгоритма заполнение таблицы

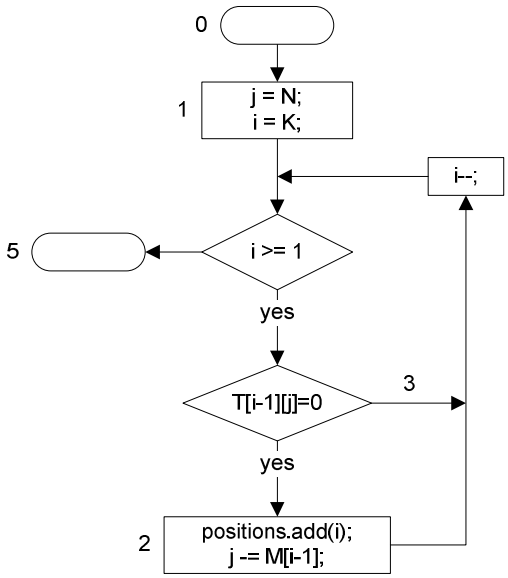

Рис. 43. Схема алгоритма поиска результатов

9. Преобразование схем алгоритма в графы переходов системы автоматов Мура

В соответствии с методом, изложенным в работе [58], определим состояния, которые будут иметь автоматы Мура, соответствующие схемам алгоритма. Для этого присвоим номера последовательностям операторных вершин (последовательность может состоять и из одной вершины).

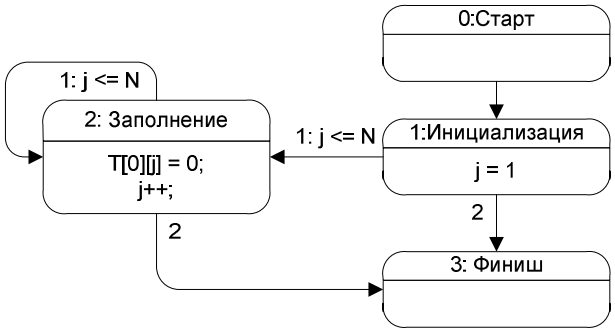

Рис. 44. Автомат *aFR*, реализующий заполнение первой строки

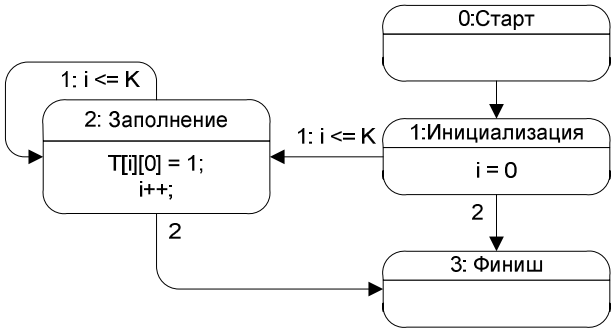

Рис. 45. Автомат *aFC*, реализующий заполнение первого столбца

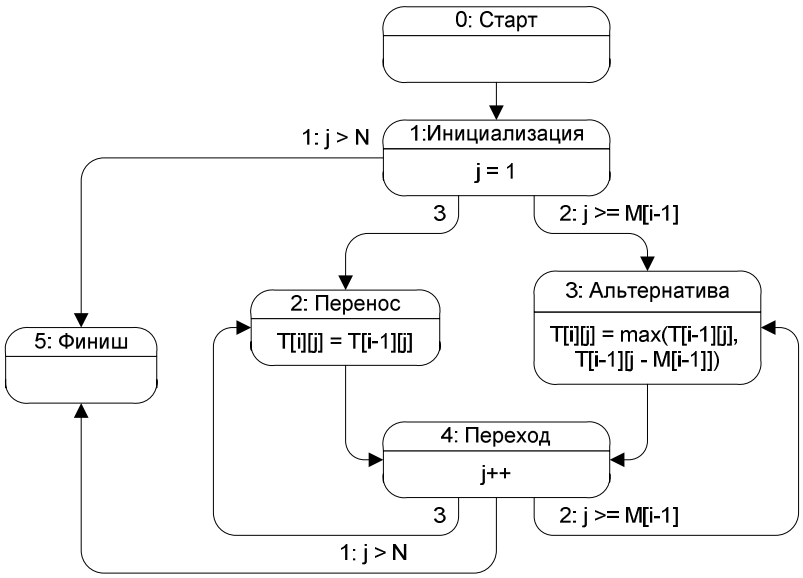

Рис. 46. Автомат *aTR*, реализующий заполнение одной строки таблицы

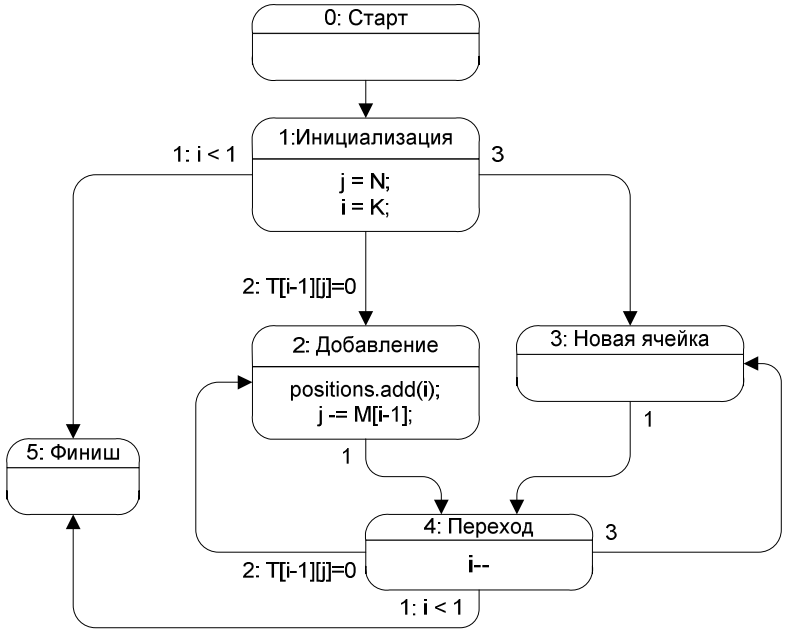

Рис. 47. Автомат *aR*, реализующий поиск результата

Отметим, что если автоматы на рис. 44-47 получены непосредственно с помощью метода, изложенного в работе [58], то остальные автоматы требуют также использования преобразования, приведенного на рис. 35. При использовании этого преобразования, парные состояния, соответствующие вызову вложенной процедуры, будут иметь состояния *S* и *S′*.

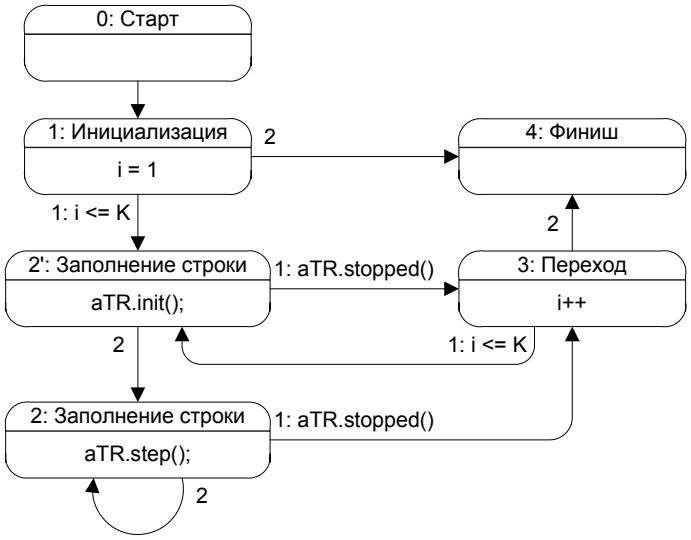

Рис. 48. Автомат *aT*, реализующий заполнение таблицы

Автомат, представленный на рис. 48, использует в качестве вложенного автомат *aTR*, представленный на рис. 46. На рис. 49 представлен главный автомат *aM*, реализующий алгоритм решения «задачи о рюкзаке». Отметим, что поскольку в данном примере, как и во многих других, при преобразовании схемы алгоритма, приведенного на рис. 43 необходимо выделить дополнительное состояние 3 (рис. 47). Несмотря на то, что в этом состоянии не производится никаких действий, оно необходимо для демонстрации важного аспекта алгоритма. Иллюстрация в состоянии 3 показывает ситуацию перехода к предыдущей строке таблицы *T*, когда текущий элемент не входит в результирующий набор.

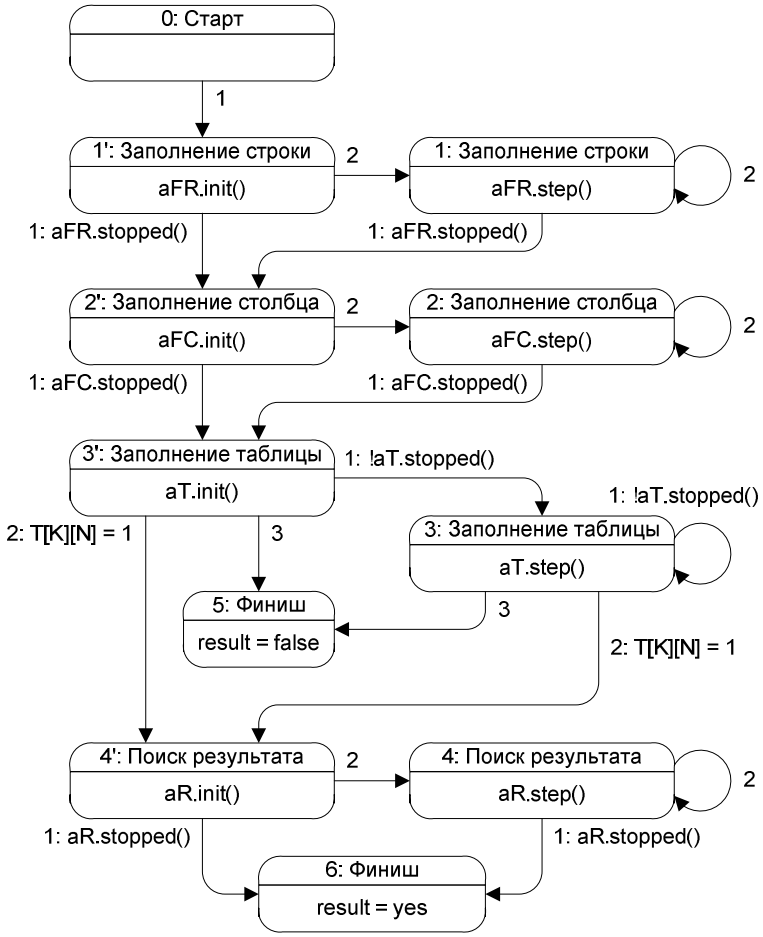

Рис. 49. Главный автомат *aM*, реализующий решение «задачи о рюкзаке»

10. Формирование набора невизуализируемых состояний для каждого автомата

Из анализа графов переходов следует, что «интересными» (изменяющими визуализацию) являются следующие состояния:

- Для автомата *aFR*, реализующего заполнение первой строки таблицы (рис. 44), интересным является только состояние 2, в котором происходит заполнение ячейки значением 0.
- Автомат *aFC*, реализующий заполнение первого столбца (рис. 45), также имеет только одно интересное состояние – состояние 2.
- В автомате *aTR*, реализующем заполнение одной строки таблицы (рис. 46), интересными являются состояния 2 и 3, отображающие две различные альтернативы при заполнении значения в текущей ячейке таблицы *T*.

- Автомат *aT*, реализующий заполнение таблицы *T* (рис. 48), не имеет интересных состояний. При этом, однако, пошаговое выполнение алгоритма визуализатором останавливается в состояниях 2 и 2', поскольку автомат *aTR*, вложенный в автомат *aT*, имеет интересные состояния.
- В автомате *aR*, реализующем поиск результата (рис. 47), все состояния, кроме конечного состояния 5, являются интересными.
- В главном автомате (рис. 49) интересными являются состояния 0, 5 и 6.

Как было отмечено, наличие интересных состояний во вложенном автомате приводит к тому, что визуализатор будет останавливаться и в тех состояниях автомата, которые взаимодействуют с вложенным автоматом.

11. Назначение уровня детализации каждому из автоматов

Анализ алгоритма приводит к тому, что уровни детализации автоматов назначаются следующим образом (рис. 50).

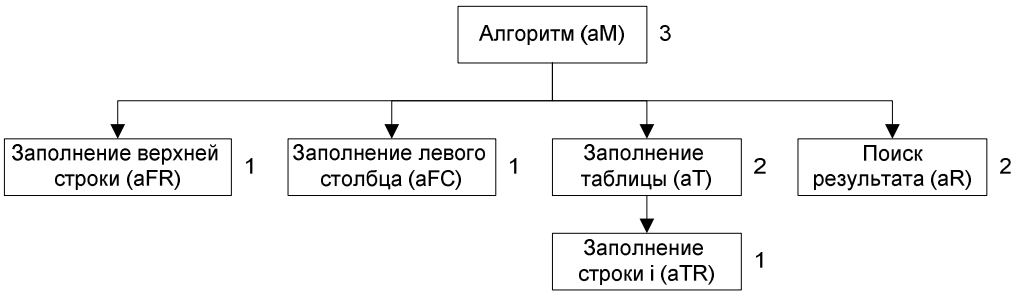

Рис. 50. Уровни детализации автоматов

Поскольку заполнение верхней строки, первого столбца и любой строки таблицы *T* является простым повторяющимся действием, то автоматы, реализующие эти действия, имеют первый уровень детализации (самый высокий уровень детализации). Поэтому при нажатии на клавишу пропуска детализации первого уровня, заполнение верхней строки, первого столбца, а также каждой строки таблицы, будут представляться одним шагом.

Несмотря на то, что поиск результата является также однообразным повторяющимся действием, в силу своей важности в процессе понимания

алгоритма он отнесен ко второму уровню детализации вместе с автоматом, заполняющим таблицу.

На третьем уровне детализации находится лишь главный автомат. Поэтому при проходе по третьему уровню детализации учащийся увидит лишь следующие шаги алгоритма:

1. Заполнение верхней строки таблицы *T*.
2. Заполнение первого столбца таблицы *T*.
3. Заполнение таблицы *T*.
4. Установление факта того, что искомый набор существует.
5. Поиск искомого набора предметов.

Таким образом, наличие именно этих трех уровней детализации дает возможность учащемуся изучать алгоритм на уровнях полной и частичной детализации, а также просмотреть алгоритм на уровне крупных этапов.

12. Выбор интерфейса визуализатора

Как и ранее в настоящей работе, в верхней части визуализатора представляется следующая информация:

- таблица значений функции $T(i, j)$, где $i = 0,\ldots, 5$; $j = 0,\ldots,16$;
- набор весов предметов (крайний левый столбец с цифрами 4, 5, 3, 7, 6).

В нижней части визуализатора расположена панель управления, которая содержит следующие кнопки:

- «>» – сделать шаг алгоритма;
- «>>» – сделать шаг алгоритма, пропуская все переходы первого уровня детализации;
- «>>>» – сделать шаг алгоритма, пропуская все переходы первого и второго уровня детализации;
- «Заново» – начать алгоритм заново;
- «Авто» – перейти в автоматический режим;

- «<», «>» – изменение паузы между автоматическими шагами алгоритма.

Среднюю часть визуализатора занимает область комментариев, в которой на каждом шаге отображается номер состояния и описание действия, выполняемого в этом состоянии.

Визуализатор в исходном состоянии, которое соответствует начальному состоянию автомата, представлен на рис. 51.

Рис. 51. Начальное состояние визуализатора

13. Сопоставление иллюстраций и комментариев с состояниями автоматов

Как и ранее, поскольку при построении визуализатора используется система взаимодействующих автоматов Мура, то иллюстрации будем формировать в системе рассматриваемых состоянии, отображая в них выполняемые действия.

Состояние 0 главного автомата *aM* (рис. 49) представлено на рис. 51.

В **состояниях 1 и 1'** главного автомата *aM* работает вложенный автомат *aFR* (рис. 44), который имеет только одно интересное **состояние 2.** В этом состоянии отображается заполнение нулями первой строки таблицы *T* (рис. 52).

Рис. 52. Состояние 2 автомата *aFR* при состоянии 1 или 1' автомата *aM*

В **состояниях 2 и 2'** главного автомата *aM* работает вложенный автомат *aFC* (рис. 45). В этом автомате также присутствует только одно интересное состояние 2, в котором отображается заполнение единицами первого столбца таблицы *T* (рис. 53).

Рис. 53. Состояние 2 автомата *aFC* при состоянии 2 или 2' автомата *aM*

В **состояниях 3 и 3'** главного автомата *aM* (рис. 49) работает вложенный автомат *aT* (рис. 48), отвечающий за заполнение таблицы *T* значениями. Как отмечено выше, автомат *aT* не имеет интересных состояний. Однако в **состояниях 2 и 2'** автомата *aT* работает вложенный автомат *aTR* (рис. 46), который имеет интересные состояния. Наличие интересных состояний в автомате *aTR* делает состояния 2 и 2' автомата *aT* также визуализируемыми, что в свою очередь делает визуализируемыми состояния 3 и 3' главного автомата *aM*.

Автомат *aTR* имеет два интересных состояния. В **состоянии 2** (рис. 54) отображается перенос значения из предыдущей строки, в то время как в **состоянии 3** (рис. 55) отображается появление альтернативного варианта. Тем

самым значение в текущей ячейке заполняется большим из альтернативных значений.

Рис. 54. Состояние 2 автомата *aTR* при состоянии 2 или 2' автомата *aT* и состоянии 3 или 3' автомата *aM*

Рис. 55. Состояние 3 автомата *aTR* при состоянии 2 или 2' автомата *aT* и состоянии 3 или 3' автомата *aM*

В **состояниях 4 и 4'** главного автомата *aM* работает вложенный автомат *aR* (рис. 47), реализующий поиск результирующего набора. Как отмечалось выше, у этого автомата интересными являются пять состояний. Рассмотрим каждое из интересных состояний в отдельности.

Состояние 0 автомата *aR* отображает факт существования решения (рис. 56).

Рис. 56. Состояние 0 автомата *aR* при состоянии 4 или 4' автомата *aM*

Состояния 1 и 4 автомата *aR* отображают выбор альтернативных вариантов при обратном проходе по таблице T. При этом состояние 1 отображает эту альтернативу на первом шаге обратного прохода таблицы, а состояние 4 – при последующих шагах (рис. 57).

Рис. 57. Состояние 4 автомата *aR* при состоянии 4 или 4' автомата *aM*

Состояния 2 и 3 автомата *aR* отображают результат выбора альтернативных вариантов при обратном проходе по таблице *T*. В состоянии 2 отображается ситуация, в которой следующий предмет найден, в то время как в состоянии 3 отображается ситуация, в которой происходит переход к следующей ячейке (рис. 58–59).

Рис. 58. Состояние 2 автомата *aR* при состоянии 4 или 4' автомата *aM*

Рис. 59. Состояние 3 автомата *aR* при состоянии 4 или 4' автомата *aM*

На рис. 60 изображено **состояние 5** главного автомата *aM*, соответствующее случаю, в котором решение отсутствует.

Рис. 60. Состояние 5 главного автомата *aM*

На рис. 61 изображено **состояние 6** главного автомата *aM*, соответствующее случаю, в котором решение найдено.

Рис. 61. Состояние 6 главного автомата *aM*

14. Выбор архитектуры программы визуализатора

Архитектура визуализатора выбрана аналогично базовому методу. Она построена на основе паттерна *MVC* [52]. При этом полученные на предыдущих шагах автоматы Мура реализуют контроллер, поскольку они обрабатывают глобальные данные, представляющие модель (рис. 62).

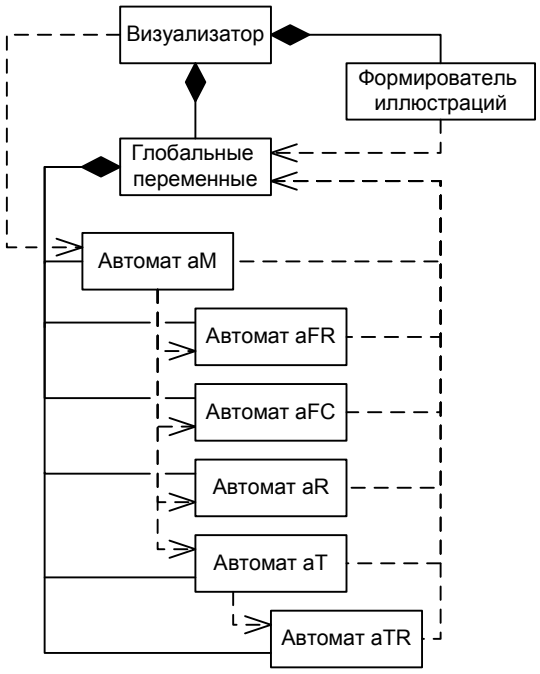

Рис. 62. Архитектура визуализатора

Как видно из схемы взаимосвязей компонент визуализатора, он воздействует лишь на главный автомат алгоритма. Система взаимодействующих автоматов самостоятельно обеспечивает не только логику алгоритма, но также логику разбиения на уровни визуализации.

15. Программная реализация визуализатора

Поскольку в рассматриваемом примере используются автоматы Мура, то программная реализация каждого автомата выполняется с помощью двух операторов *switch*, первый из которых реализует переходы между состояниями, а второй – действия в них. Как было отмечено выше, формирователь иллюстраций реализуется с помощью вложенных операторов *switch* (листинг 7).

Листинг 7. Реализация формирователя иллюстраций для системы взаимодействующих автоматов

```
1   switch (globals.aM.state()) {
2     case 0:
3       // Иллюстрация: начальное состояние
4     case 11: // в состояниях 1′ и 1 иллюстрации совпадают
5     case 1:
6       switch (globals.aFR.state()) {
7         case 2:
8           // Иллюстрация: заполнение первой строки
9       }
10    case 21: // в состояниях 2′ и 2 иллюстрации совпадают
11    case 2:
12      switch (globals.aFC.state()) {
13        case 2:
14          // Иллюстрация: заполнение первого столбца
15      }
16    case 31: // в состояниях 3′ и 3 иллюстрации совпадают
17    case 3:
18      switch (globals.aT.state()) {
19        case 21: // в состояниях 2′ и 2 иллюстрации совпадают
20        case 2:
21          switch (globals.aTR.state()) {
22            case 2:
23              // Иллюстрация: перенос значения из предыдущей строки
24            case 3:
25              // Иллюстрация: альтернатива – выбор большего из значений
26          }
27      }
28    case 41: // в состояниях 4′ и 4 иллюстрации совпадают
29    case 4:
30      switch (globals.aR.state()) {
31        case 0:
32          // Иллюстрация: набор существует
33        case 1: // в состояниях 1 и 4 иллюстрации совпадают
34        case 4:
35          // Иллюстрация: альтернатива обратного прохода при поиске набора
36        case 2:
37          // Иллюстрация: новый предмет найден
38        case 3:
39          // Иллюстрация: переход к предыдущей строке
40      }
41    case 5:
42      // Иллюстрация: набор не существует
43    case 6:
44      // Иллюстрация: конец, набор найден
45  }
```

Для пропуска неинтересных состояний используется следующий код (листинг 8).

Листинг 8. Реализация шага визуализации с учетом пропуска неинтересных, а также детализированных переходов

```
1   /**
2    * Один шаг визуализатора состоит из нескольких переходов автомата
3    */
4   public final void step() {
5     // осуществить автомата (если есть вложенный автомат,
6     // то переход будет осуществлен у вложенного автомата)
7     makeAutomationStep();
8     // пропустить неинтересные детальные состояния
9     skipHiddenStates();
10  }
11  /**
12   * Пропуск неинтересных состояний
13   */
14  private void skipHiddenStates() {
15    // если состояние скрытое либо текущий автомат находится ниже уровня пропуска
16    while (!stopped() && (isHidden() || level < g.skipLevel)) {
17      // при завершении вложенного автомата – убрать на него ссылку
18      if (inner != null && inner.stopped()) {
19        inner = null;
20      }
21      // если текущий автомат находится выше уровня пропуска,
22      // то пропуск надо завершить
23      if (g.skipLevel <= level) {
24        g.skipLevel = 0;
25      }
26      // осуществить еще один переход автомата
27      makeAutomationStep();
28    }
29  }
30  /**
31   * Возвращает значение истины, если состояние надо пропустить
32   */
33  private boolean isHidden() {
34    // скрытым является состояние, в котором нет вложенных автоматов
35    // с видимым состоянием и оно включено в список неинтересных состояний
36    return (inner != null && inner.isHidden() || inner == null)
37            && Arrays.binarySearch(hiddenStates, state) >= 0;
38  }
```

3.3. Сравнение с базовым методом

Следует отметить, что выделение уровней детализации не только делает визуализаторы алгоритмов более удобными в восприятии, но и улучшает понимание алгоритма учащимся, реализующим визуализатор [77]. Понимание достигается посредством необходимости более глубокого понимания сути алгоритма при выделении уровней детализации.

Несмотря на то, что система взаимодействующих автоматов реализует тот же алгоритм, что и один автомат, число состояний существенно увеличивается, что приводит к большему коду (табл. 5).

Таблица 5. Сравнение метрик автоматов для примера «задача о рюкзаке»

Особенность	**Один автомат**	**Система автоматов**
Число автоматов	1	6
Общее число состояний автоматов	16	32
Число строк кода для реализации автоматов	187	455

Дальнейшее развитие метода изложенного в третьей главе реализовано в статье [74], а также в системе визуализации *Vizi* [42, 43].

Выводы по главе 3

1. В главе изложен формализованный метод построения визуализаторов алгоритмов на основе системы взаимодействующих автоматов.
2. Предложенный метод не меняет поведения визуализатора при использовании пошагового режима. Таким образом, предложенный метод является расширением базового метода.
3. Предложенный метод реализует две важные возможности визуализаторов. Во-первых, он предоставляет возможность пропускать детали алгоритма, а, во-вторых, позволяет выделить в алгоритме уровни детализации.
4. Наличие дополнительных возможностей в визуализаторах, построенных предложенным методом, требует построения большего числа автоматов с большим числом состояний. Следствием из этого является более длинный код, реализующий логику визуализатора.

ГЛАВА 4. АВТОМАТНЫЙ ПОДХОД К ОБЕСПЕЧЕНИЮ ПРОСТОЙ АНИМАЦИИ В ВИЗУАЛИЗАТОРАХ АЛГОРИТМОВ ДИСКРЕТНОЙ МАТЕМАТИКИ

Существуют различные виды и методы анимации. Для визуализаторов алгоритмов под простой анимацией будем понимать обеспечение непрерывного (без скачков) перехода алгоритма из одной вершины в другую.

Ранее визуализатор рассматривался, как дискретная последовательность статических иллюстраций, что в большинстве случаев достаточно для обучения. Однако, в некоторых алгоритмах статических иллюстраций мало. Примерами таких алгоритмов могут служить пирамидальная сортировка (сортировка с помощью кучи) или обход дерева в глубину [2]. В пирамидальной сортировке процессы передвижения значений между вершинами дерева, в виде которого представлен массив, наиболее наглядно демонстрируется при помощи анимации, а в алгоритме обхода дерева в глубину именно процесс движения указателя по вершинам дерева является наиболее интересным и важным для понимания этого алгоритма.

В настоящей главе предлагается расширение метода построения визуализаторов с целью включения в нее возможности анимации требуемых шагов визуализации. При этом рассматривается такая разновидность анимации, при которой изображается переход от предыдущих значений визуализируемых переменных к следующим. Поскольку каждая статическая иллюстрация является функцией от визуализируемых переменных, то указанная анимация и будет визуализировать соответствующий шаг алгоритма в динамике. Предлагаемый подход иллюстрируется на двух примерах: традиционной реализации алгоритма пирамидальной сортировки и алгоритме обхода дерева в глубину.

4.1. Метод обеспечения простой анимации

С целью обеспечения возможности анимации в последовательность действий, приведенную во второй главе, вводятся дополнительные шаги. Отметим, что поскольку изменения визуализируемых переменных в используемых автоматах Мура производятся в состояниях (что весьма удобно), то будем рассматривать вопрос о введении простой анимации в визуализаторы, построенные на основе таких автоматов. В виду того, что при введении анимации в автомате визуализации появляются действия на дугах, то этот автомат становится смешанным.

Для построения визуализатора с анимацией предлагается применять четыре автомата:

- *автомат, реализующий алгоритм* (формируется на восьмом этапе метода, указанном выше);
- *автомат визуализации* (формируется на девятом этапе);
- *преобразованный автомат визуализации*;
- *автомат анимации*.

Автомат визуализации отображает последовательно *статические иллюстрации* в каждом состоянии, что приводит к *статической (пошаговой) визуализации*. Под статической иллюстрацией понимается фиксированный кадр, не изменяющийся с течением времени.

Преобразованный автомат визуализации, кроме статических иллюстраций, отображает также и *динамические иллюстрации* в дополнительно вводимых анимационных состояниях. Это позволяет говорить о *динамической визуализации (анимации)*.

Динамическая иллюстрация состоит из набора промежуточных статических иллюстраций, отображаемых на экране *автоматом анимации* через определенные промежутки времени. Это приводит к динамическому изменению иллюстрации в анимационном состоянии.

4.1.1. Дополнительные этапы метода для обеспечения анимации

Для автоматов Мура анимацию естественно проводить (также как и формирование статических изображений и комментариев) в состояниях. Для введения анимации в визуализатор расширим метод за счет введения следующих шагов:

a) **выбор состояний автомата визуализации, в которых выполняется анимация** (такие состояния будем назвать *анимационными*);

b) **построение преобразованного автомата визуализации путем замены каждого из анимационных состояний тремя состояниями**, в первом из которых производятся преобразования данных, во втором – анимация, соответствующая этому состоянию, при переходе в третье состояние анимация заканчивается, а непосредственно в третьем состоянии отображается статическая иллюстрация, соответствующая состоянию исходного автомата;

c) **разработка анимационного автомата и обеспечение его взаимодействия с преобразованным автоматом визуализации** (выполняется один раз для всех визуализаторов, проектируемых с помощью этого метода);

d) **разработка и реализация функции вывода каждой динамической иллюстрации, зависящей от состояния автомата визуализации и шага анимации**.

4.1.2. Формализация построения автомата визуализации

До перехода к более подробному рассмотрению новых этапов изложим, в чем состоит девятый этап исходного метода (рис. 63).

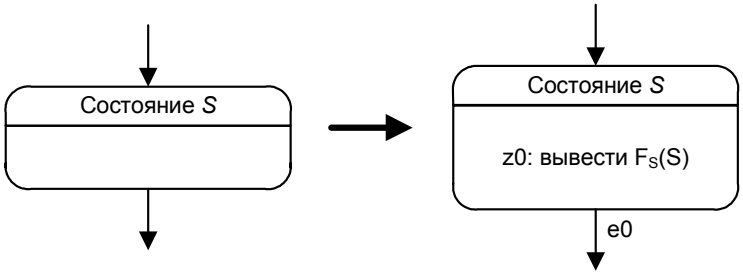

Рис. 63. Формирование состояний автомата визуализатора

На этом этапе вводится выходное воздействие *z0*, осуществляющее формирование статического изображения и комментария в рассматриваемом состоянии. Различия в изображениях и комментариях формализуются с помощью функции $F_S(S)$, параметром которой является номер состояния *S*. В каждом состоянии, формирующем выходное воздействие *z0*, переход к следующему состоянию выполняется по событию *e0* (нажатие клавиши «шаг» в интерфейсе визуализатора).

4.1.3. Введение анимационных состояний в автомат визуализации

Поясним каждый из вновь вводимых этапов *a* – *e*.

На этапе *a* выполняется выбор анимационных состояний в соответствии с особенностями алгоритма и принятыми решениями по анимации на четвертом этапе исходного метода («Анализ алгоритма для визуализации»). Так как в алгоритме обхода двоичного дерева наибольший интерес представляет процесс перехода между вершинами, то состояния, в которых происходит переход, и будут выбраны в качестве анимационных.

На этапе *b* в автомате визуализации для получения преобразованного автомата этого типа каждое анимационное состояние формально заменяется тремя состояниями (рис. 64).

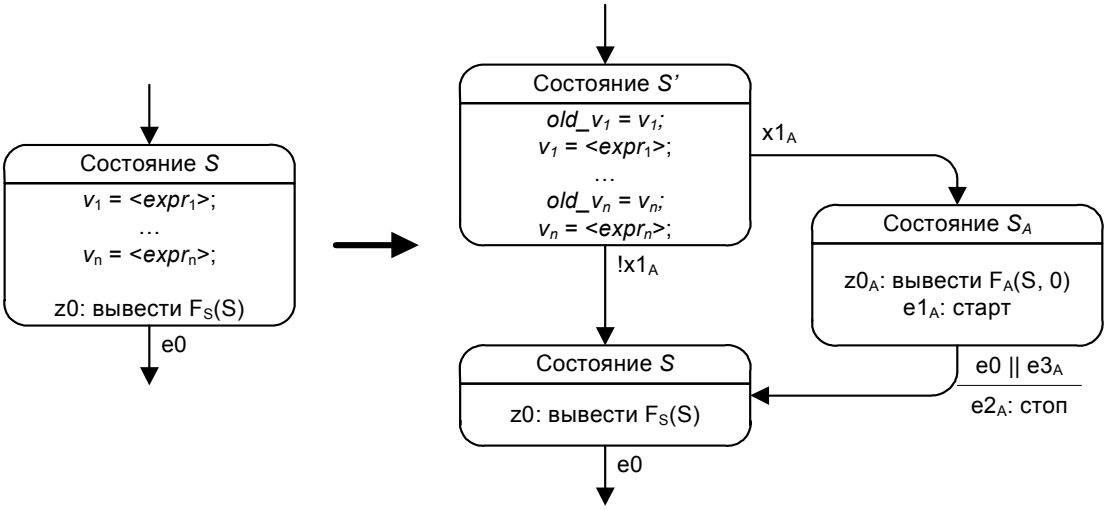

Рис. 64. Преобразование анимационного состояния

Опишем это преобразование. Предположим, что состояние S выбрано для визуализации. Предположим также, что в этом состоянии изменяются значения визуализируемых переменных $v_1, \ldots, v_n$. При этом переменной v_i присваивается значение выражения $< expr_i >$. Преобразование состоит в замене выбранного состояния на три состояния – S', S и S_A.

В состоянии S' значения переменных $v_1, \ldots, v_n$ предварительно сохраняются в переменных $old\_v_1, \ldots, old\_v_n$, а также проводятся все действия, выполняемые в состоянии S автомата визуализации.

В состоянии S' преобразованного автомата визуализации в отличие от состояния S исходного автомата этого типа не происходит ожидания события, а выполняется переход в одно из состояний S_A или S.

Переход в состояние S_A производится, если переменная $x1_A$ принимает значение истина (анимация включена). Анимация осуществляется, пока преобразованный автомат визуализации находится в состоянии S_A. При входе в это состояние выполняется запуск автомата анимации при помощи события $e1_A$: *старт* и отображается иллюстрация $F_A(S, 0)$, соответствующая началу анимации.

Выход из состояния анимации и переход в состояние S происходит, как по событию $e0$ (нажатие клавиши), так и по событию $e3_A$ (окончание анимации), и сопровождается сигналом $e2_A$: *стоп*. Таким образом,

обеспечивается как автоматическое, так и ручное завершение анимации. Схема взаимодействия [59] преобразованного автомата визуализации представлена на рис. 65.

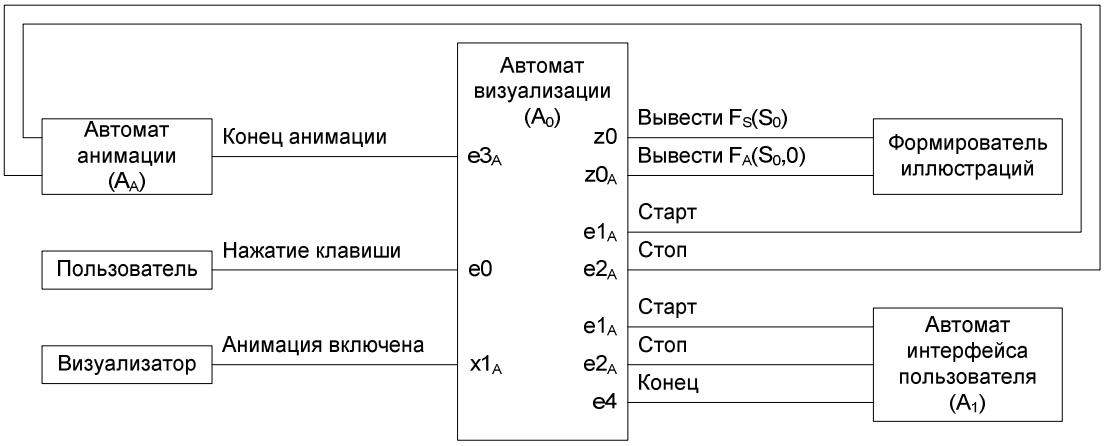

Рис. 65. Схема взаимодействия преобразованного автомата визуализации

Состояние *S* является завершающим. В нем выполняется отображение статической иллюстрации $F_S\ (S)$. Отметим, что поскольку в преобразованном анимационном автомате присутствуют действия на дугах, то этот автомат является *смешанным автоматом – автоматом Мура-Мили*.

Следует отметить, что при выключении анимации (флаг $x1_A$ принимает значение ложь) процесс работы преобразованного автомата визуализации полностью повторяет исходный автомат визуализации. Поэтому включение анимации может рассматриваться именно как расширение исходного визуализатора.

4.1.4. Построение автомата анимации

На этапе *c* строится автомат анимации (рис. 66). Следует отметить, что он является смешанным автоматом, поскольку содержит действия как на дугах, так и в вершинах графа перехода.

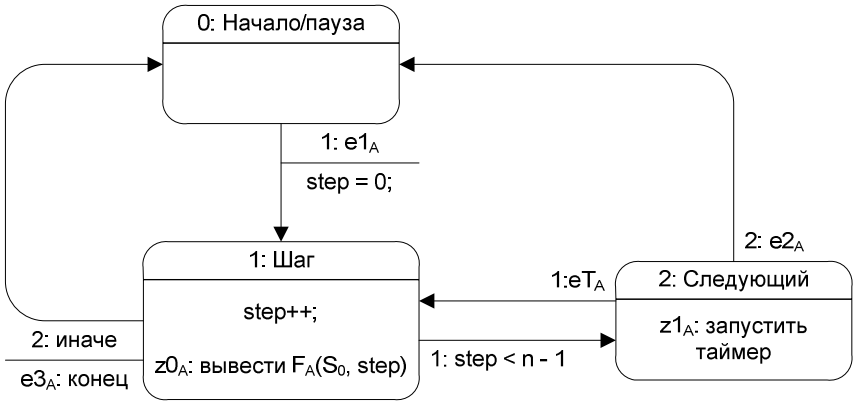

Рис. 66. Автомат анимации

Из рассмотрения этого рисунка следует, что автомат предназначен для изменения значения переменной *step*, которая хранит номер шага анимации.

Автомат анимации формирует следующие выходные воздействия и события:

- $z1_A$*: запустить таймер* – запускает таймер, который через период времени *T* генерирует событие eT_A;
- $z0_A$*: вывести* $F_A(S_0, step)$ – информирует «рисовальщик» о необходимости перерисовать иллюстрацию;
- $e3_A$*: конец анимации* – сигнализирует о том, что анимация закончилась.

Анимационный автомат реагирует на следующие события:

- $e1_A$: *старт* – автомат визуализации перешел в состояние, в котором должна начаться анимация;
- $e2_A$: *стоп* – автомат визуализации перешел в состояние, в котором анимация должна закончиться;
- eT_A – событие от таймера, по которому осуществляется переход к следующему шагу анимации.

Описанный автомат является универсальным и может быть использован в любом визуализаторе.

На этапе *d* автомат визуализации и автомат анимации связываются посредством событий и выходных воздействий, как показано на рис. 67.

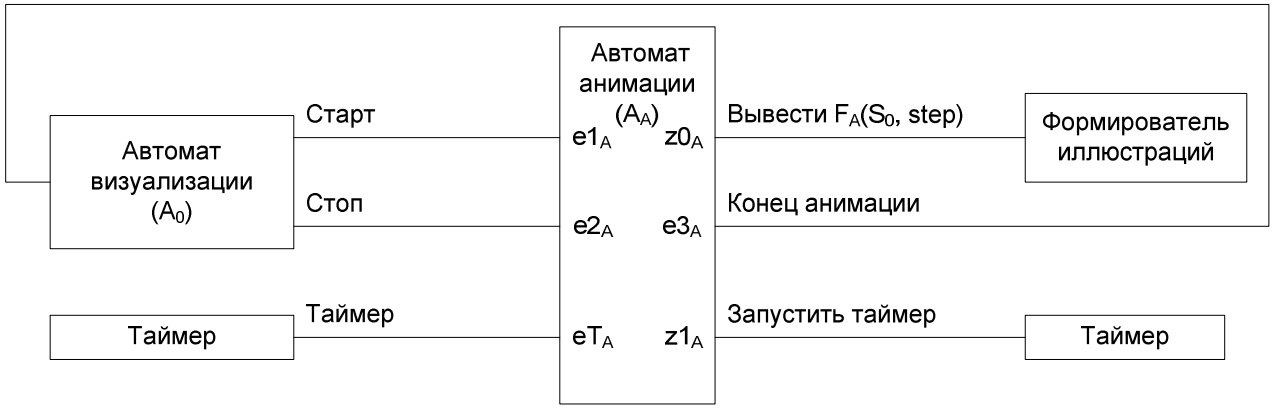

Рис. 67. Схема связей автомата анимации

4.1.5. Обеспечение отображения анимации

На этапе *e* стандартная функциональность «рисовальщика» сопоставления номера состояния с текстовым комментарием и статической иллюстрацией расширяется. В режиме анимации «рисовальщик» принимает на вход значение номера шага. Другими словами, если функция F_S реализует статические иллюстрации, а функция F_A – динамические (анимационные) иллюстрации, то при переходе от статики к анимации осуществляется преобразование вида:

$$F(state) \to \begin{cases} F_S(state); \\ F_A(state, step) \end{cases}$$

При этом справедливы следующие соотношения:

$$F_S(state) = F(state) \quad (1)$$

$$F_A(state, 0) = F_S(state - 1) \quad (2)$$

$$F_A(state, N) = F_S(state) \quad (3)$$

Соотношение (1) отражает тот факт, что преобразованный автомат визуализации формирует статические иллюстрации, совпадающие с иллюстрациями исходного автомата. Соотношения (2) и (3) показывают, что формирование динамических (анимационных) иллюстраций в состоянии *state* происходит таким образом, что начальная анимационная иллюстрация совпадает со статической иллюстрацией в предыдущем состоянии автомата, а конечная – со статической в состоянии *state*.

4.1.6. Введение новых этапов в метод

Из изложенного следует, что рисовальщик в визуализаторе в общем случае будет реализовываться посредством двух операторов *switch*. При этом один оператор *switch* будет соответствовать статическим иллюстрациям, а другой – динамическим.

Таким образом, первые четыре перечисленных шага вводятся в указанный выше метод между этапами с номерами девять и десять, а пятый шаг – между этапами одиннадцать и двенадцать. При этом этапы базового метода преобразуются следующим образом:

1. **Постановка задачи**.
2. **Решение задачи** (в словесно-математической форме).
3. **Выбор визуализируемых переменных**.
4. **Анализ алгоритма для визуализации**. Анализируется решение с целью определения того, что и как отображать на экране.
5. **Реализация алгоритма решения задачи**.
6. **Реализация алгоритма на выбранном языке программирования**. На этом шаге производится реализация алгоритма, его отладка и проверка работоспособности.
7. **Построение схемы алгоритма по программе**.
8. **Преобразование схемы алгоритма в граф переходов автомата Мили либо Мура**.
9. **Формирование набора невизуализируемых переходов (состояний)**.
10. **Выбор состояний, в которых будет выполняться анимация**.
11. **Замена каждого из выбранных состояний тремя состояниями**.
12. **Использование автомата анимации и обеспечение его взаимодействия с преобразованным автоматом визуализации**.
13. **Выбор интерфейса визуализатора**.

14. **Сопоставление иллюстраций и комментариев с состояниями автомата**.
15. **Обеспечение выбора в каждом анимационном состоянии статического либо динамического изображения**, зависящего не только от состояния основного автомата, но и от номера шага анимации.
16. **Выбор архитектуры программы визуализатора**.
17. **Программная реализация визуализатора**.

Продемонстрируем работу метода на двух примерах широко известных алгоритмов.

4.2. Построение визуализатора на примере алгоритма пирамидальной сортировки

Построим визуализатор алгоритма пирамидальной сортировки [2]. Этот алгоритм является решением общей задачи внутренней сортировки массива.

1. Постановка задачи сортировки (в общем виде)

Задача сортировки состоит в упорядочивании последовательности записей таким образом, чтобы значения ключевого поля составляли неубывающую последовательность [5]. В настоящей работе рассматривается более конкретный случай сортировки массива целых чисел. Задан массив a, состоящий из n элементов. Требуется реализовать алгоритм, который располагает элементы массива a в неубывающем порядке.

2. Решение задачи на основе алгоритма пирамидальной сортировки

Приведем словесную формулировку решения этой задачи.

Суть алгоритма пирамидальной сортировки состоит в построении и поддержании так называемой «кучи» или «пирамиды» [2]. «Пирамида» – это двоичное дерево, значения элементов в котором следуют правилу: значение родительского элемента больше или равно значению обоих дочерних элементов.

Основной механизм для поддержки и построения пирамиды является «погружение»:

- элемент пирамиды сравнивается с обоими дочерними элементами;
- если один или оба элемента оказываются больше чем вершина, то вершина обменивается с большим из дочерних элементов;
- механизм обмена осуществляется до тех пор, пока текущая вершина не окажется больше, чем элементы в обеих дочерних вершинах.

До начала решения массив представляется в виде двоичного дерева, упакованного в массив. При этом предполагается, что элемент с номером i в качестве левого и правого дочерних узлов в дереве имеет элементы с номерами $2i + 1$ и $2i + 2$.

Решение задачи предлагается строить в два этапа:

- **на первом этапе** осуществляется первичное построение «пирамиды». Для этого осуществляется последовательное «погружение» всех элементов, начиная с элемента с номером $n / 2$ до первого включительно;
- **на втором этапе** все элементы последовательно исключаются из «пирамиды» путем перемещения наибольшего элемента, находящего в вершине «пирамиды», за ее пределы. При этом на освободившееся место наибольшего элемента перемещается элемент из конца «пирамиды». Другими словами, если *hsize* – текущий размер «пирамиды», то шаг состоит из уменьшения значения переменной *hsize* на единицу и обмена элементов с номерами 0 и *hsize*. В конце шага для поддержания «пирамиды» элемент с вершины «погружается». По окончании $(n–1)$-го шага из «пирамиды» будут исключены все элементы, кроме одного, а элементы массива будут расположены в неубывающем порядке.

3. Выбор визуализируемых переменных

Предлагается визуализировать следующие переменные:

- *array* – массив элементов;
- *c* – текущая вершина;
- *cmax* – вершина с большим элементом при сравнениях;
- *hsize* – число элементов в массиве, которые входят в «пирамиду» на данном шаге алгоритма.

4. Выбор алгоритма визуализации

Предлагается визуализировать следующие особенности алгоритма для обеспечения возможности наилучшего разъяснения его действия:

- процесс сравнения вершины с обоими дочерними;
- обмены элементов;
- оставшаяся часть пирамиды.

5. Реализация алгоритма решения задачи

Приведем алгоритм решения задачи на псевдокоде [2]:

```
1       hsize ← длина a
2       usize ← длина a / 2
3       while true do
4         if usize > 0 then
5           usize ← usize – 1
6         else if hsize > 1 then
7           hsize ← hsize – 1
8           обменять элементы 0 и hsize
9         else break
10      c = usize
11      while true do
12        cmax = наибольший из a[c], a[left(c)], a[right(c)]
13        if cmax != c then
14          обменять элементы c и cmax
15          c = cmax
16        else break
```

6. Реализация алгоритма на языке *Java*

Перепишем программу, записанную на псевдокоде, с помощью языка *Java* (листинг 9).

Листинг 9. Алгоритм пирамидальной сортировки на языке *Java*

```
1       /**
2        * Реализация алгоритма пирамидальной сортировки
3        */
4       private void solve() {
5           // Размер пирамиды в рамках массива
6           hsize = a.length;
7           // Число вершин, которые надо "погрузить"
8           usize = a.length / 2;
9           while (true) {
10              if (usize > 0) {
11                  // Первый этап - упорядочивание
12                  usize--;
13              } else if (hsize > 1){
14                  // Второй этап - обмен вершины пирамиды с последним элементом
15                  // и уменьшение размера пирамиды.
16                  hsize--;
17                  swap(0, hsize);
18              } else {
19                  // Конец. Все вершины отсортированы
20                  break;
21              }
22              // Начинаем с usize:
23              // - на первом этапе - последний "непросеянный" элемент
24              // - на втором этапе - всегда вершина (usize = 0)
25              int c = usize;
26              while (true) {
27                  // "Погружение". Выбираем максимум из элемента и дочерних
28                  int cmax = maxvalue(c, left(c), right(c));
29                  // Если элемент не является максимальным
30                  if (cmax != c) {
31                      // то обмениваем его с максимумом и продолжаем "погружение"
32                      swap(c, cmax);
33                      c = cmax;
34                  } else {
35                      // иначе переходим к следующему шагу
36                      break;
37                  }
38              }
39          }
40      }
```

В этой программе операции ввода/вывода не приведены, так как их будет выполнять визуализатор. Простейшие операции `swap()` и `maxvalue()` опущены, поскольку их реализация не является существенной.

7. Построение схемы алгоритма по программе

Построим по тексту программы схему алгоритма (рис. 68).

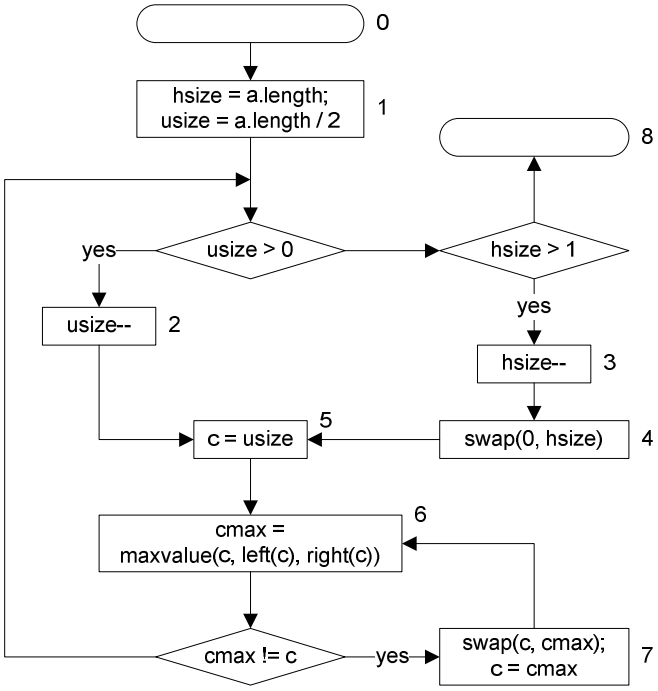

Рис. 68. Схема алгоритма пирамидальной сортировки

8. Преобразование схемы алгоритма в автомат Мура

Следуя методу, приведенному в работе [58], построим автомат Мура, соответствующий приведенной выше схеме алгоритма (рис. 69)

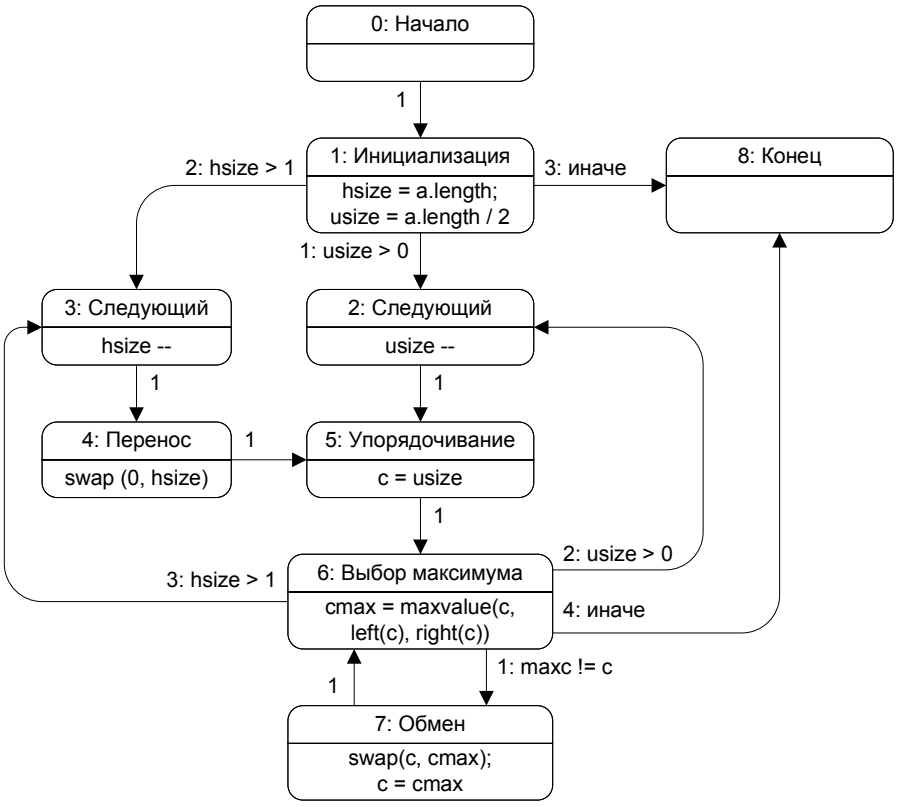

Рис. 69. Автомат Мура, реализующий алгоритм пирамидальной сортировки

Исходя из схемы алгоритма на рис. 68, выделены девять состояний. В состоянии 1 выполняется инициализация алгоритма. В состоянии 2 выполняется переход к следующему элементу на первом этапе алгоритма. В состояниях 3 и 4 элемент с вершины «пирамиды» переносится в конец и размер «пирамиды» уменьшается. Состояние 5 отвечает за выбор первой вершины для упорядочивания. В состоянии 6 выбирается максимальный элемент среди текущей вершины и дочерних, а в состоянии 7 выполняется обмен текущего элемента и максимального. Состояние 8 является конечным.

9. Преобразование автомата, реализующего алгоритм в автомат визуализации

В соответствии с преобразованием (рис. 63), по автомату (рис. 69) строится автомат визуализации (рис. 70). При этом состояние 2 не является существенным для визуализации и, как следствие не должно визуализироваться, о чем свидетельствует отсутствие ожидания события *e0* в этом состоянии.

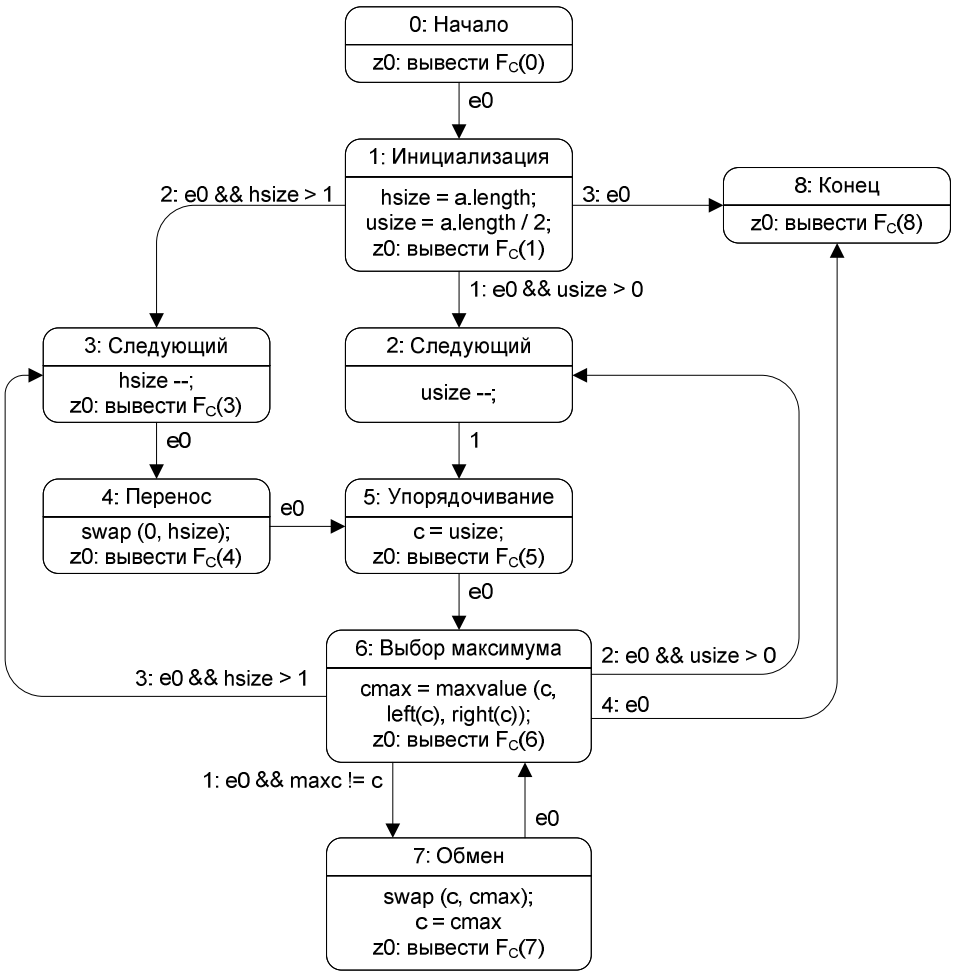

Рис. 70. Автомат визуализации

10. Выбор состояний, в которых будет выполняться анимация

В алгоритме пирамидальной сортировки наиболее важным для динамического отображения является процесс обмена значений в вершинах, что визуально представляется как движение элементов по «пирамиде». Такими состояниями являются состояния с номерами 4 и 7.

11. Замена каждого из анимационных состояний тремя состояниями

В соответствии с преобразованием (рис. 64), заменим состояния 4 и 7 тройками состояний, в первом из которых запоминается номер предыдущей вершины, во втором формируется событие начала анимации, а в третьем завершается шаг визуализатора и создается конечная иллюстрация в состоянии (рис. 71). При этом событие окончания анимации формируется на переходе между анимационным и конечным состояниями шага визуализации.

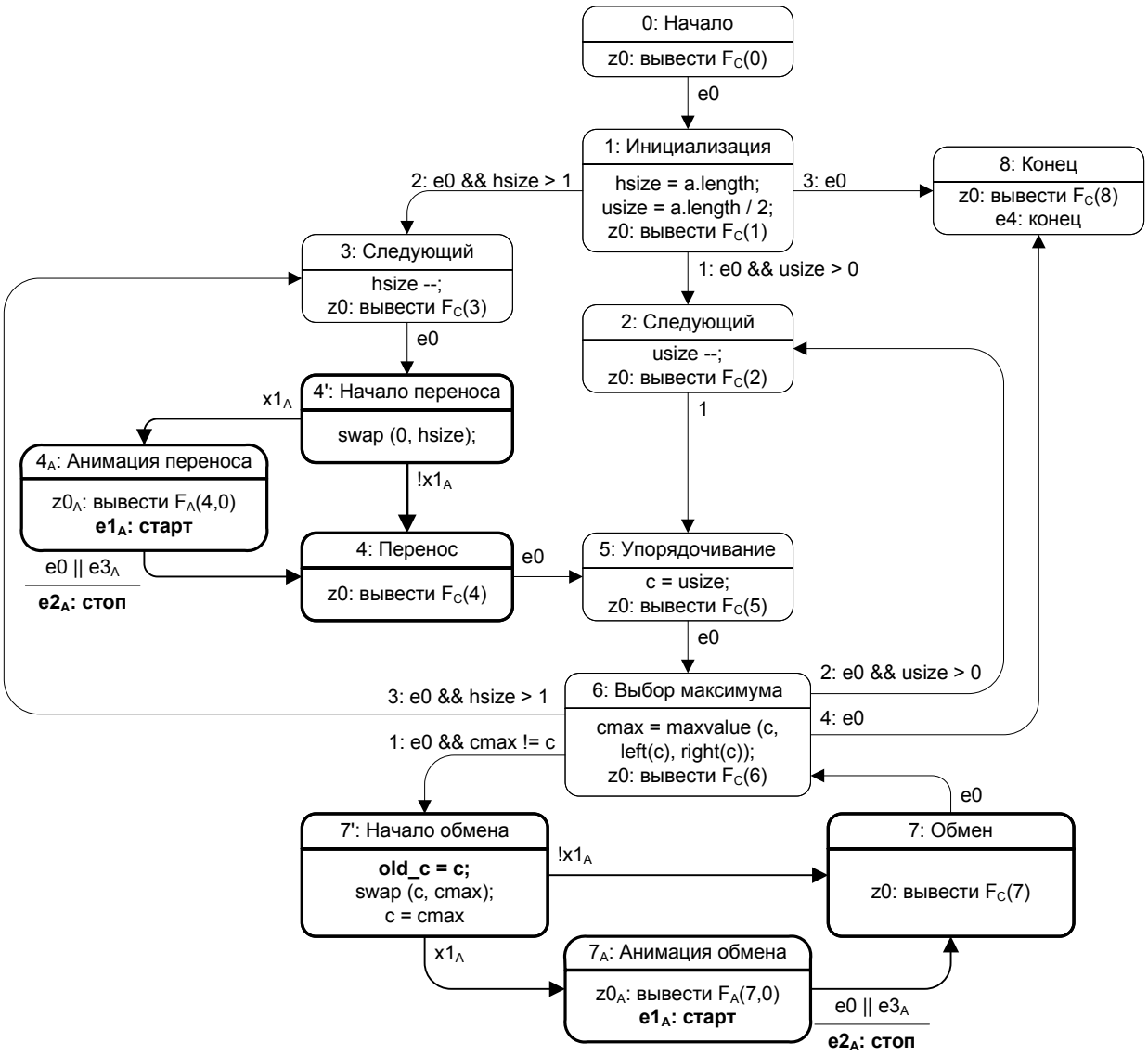

Рис. 71. Преобразованный автомат визуализации

12. Использование автомата анимации и обеспечение его взаимодействия с преобразованным автоматом визуализации

Как отмечалось выше, для анимации используется стандартный автомат рис. 66. Для обеспечения взаимодействия между автоматом визуализации и автоматом анимации вводятся связи (рис. 67).

13. Выбор интерфейса визуализатора

Визуализатор имеет интерфейс, показанный на рис. 72:

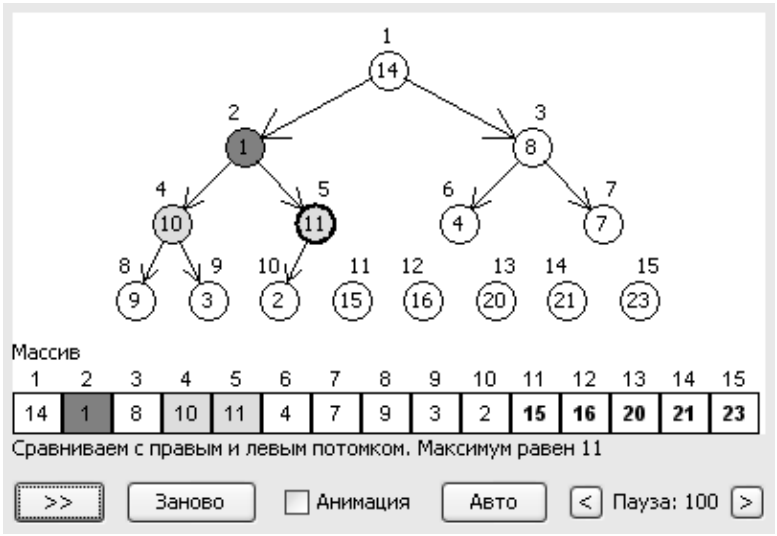

Рис. 72. Интерфейс визуализатора

- в верхней части экрана изображена «пирамида». При этом ребра изображаются лишь между элементами, входящими в текущий момент в «пирамиду»;
- темным цветом отмечена текущая вершина, а серым цветом – одна или две дочерние вершины, участвующие либо в операциях сравнения, либо обмена;
- в нижней части визуализатора находится текущее состояние массива, в котором цвета ячеек повторяют цвета соответствующих элементов в «пирамиде», что помогает следить за изменениями не только в самой «пирамиде», но и в массиве, соответствующему этой «пирамиде»;
- полужирным шрифтом в массиве отмечена отсортированная часть.

Под иллюстрацией массива динамически отображаются текстовые комментарии, соответствующие текущим состояниям автомата визуализации. Например, на рис. 72 изображен комментарий, соответствующий состоянию 6. Под комментарием расположены элементы управления. Следует отметить, что по сравнению с обычным визуализатором появляется флаг управления анимацией:

- «>>» – сделать шаг алгоритма;
- «Заново» – начать алгоритм сначала;

- «Анимация» – отображать или не отображать анимацию;
- «Авто» – перейти в автоматический режим;
- «<», «>» – изменить паузу между автоматическими шагами алгоритма.

14. Сопоставление иллюстраций и комментариев с состояниями автомата

Для наглядности визуализации алгоритма предлагается акцентировать внимание на следующих элементах.

В состоянии 0 отображается исходный массив, расположенный в виде дерева (рис. 73).

В состоянии 1 вершины связываются в «пирамиду» посредством соединения соответствующих элементов ребрами (рис. 74).

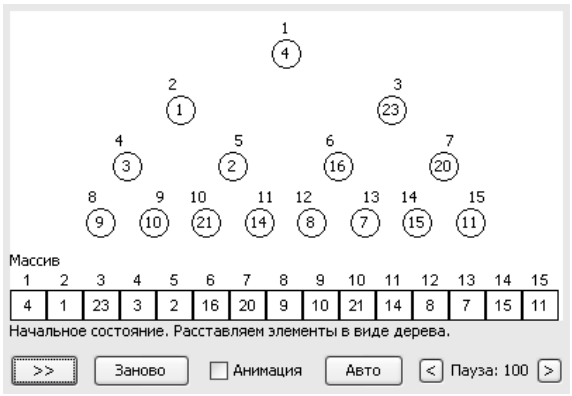

Рис. 73. Иллюстрация в состоянии 0

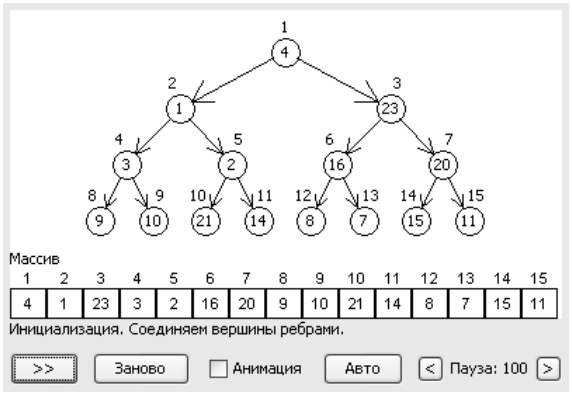

Рис. 74. Иллюстрация в состоянии 1

В состоянии 3 последний элемент массива исключается из «пирамиды» (рис. 75), а в состоянии 4 происходит обмен исключенного элемента и элемента на вершине «пирамиды» (рис. 76).

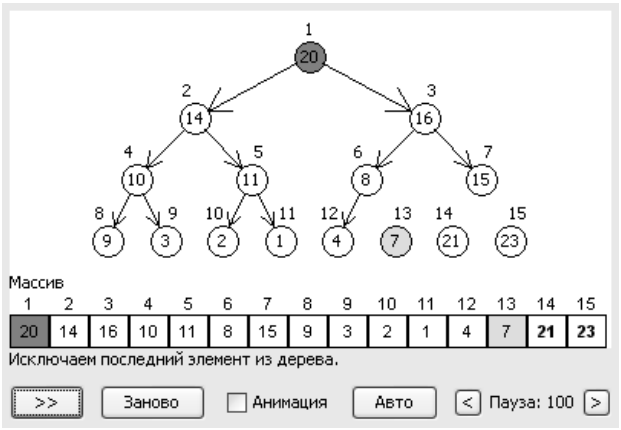

Рис. 75. Иллюстрация в состоянии 3

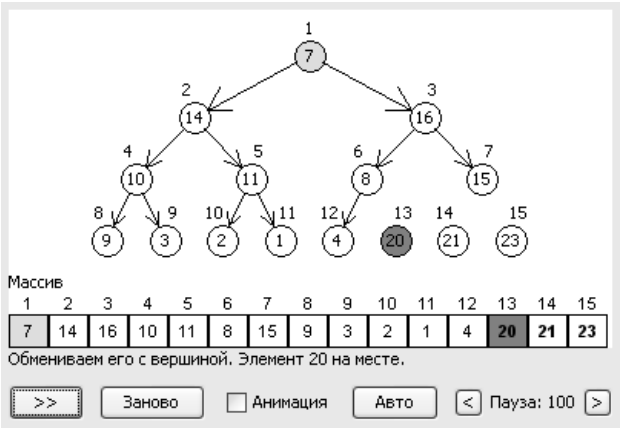

Рис. 76. Иллюстрация в состоянии 4

Состояние 5 соответствует началу «погружения» вершины (рис. 77). Состояние 6 (рис. 72) соответствует процессу сравнения элемента с правым и левым дочерними элементами, а состояние 7 соответствует обмену элемента с большим из дочерних элементов (рис. 78).

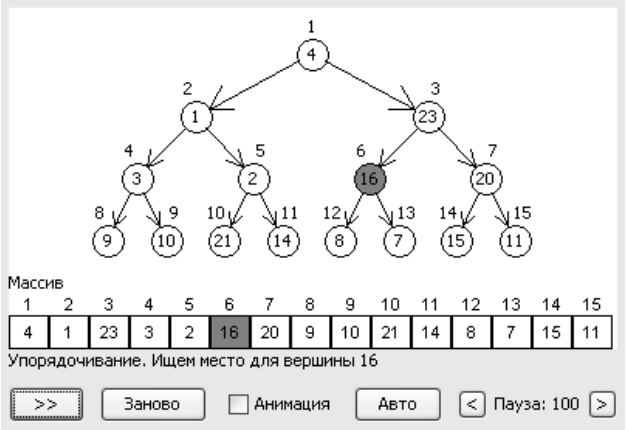

Рис. 77. Иллюстрация в состоянии 5

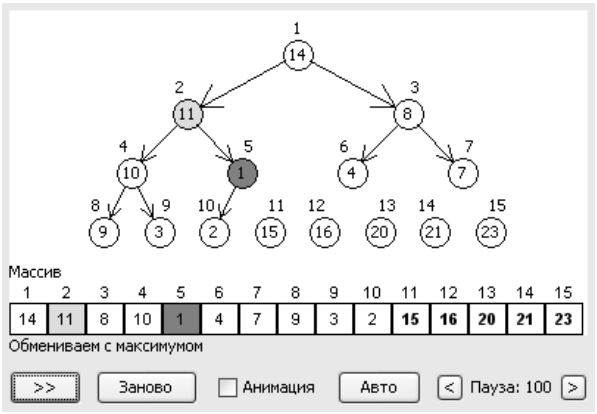

Рис. 78. Иллюстрация в состоянии 7

Состояние 8 (рис. 79) является конечным и отображает отсортированный массив.

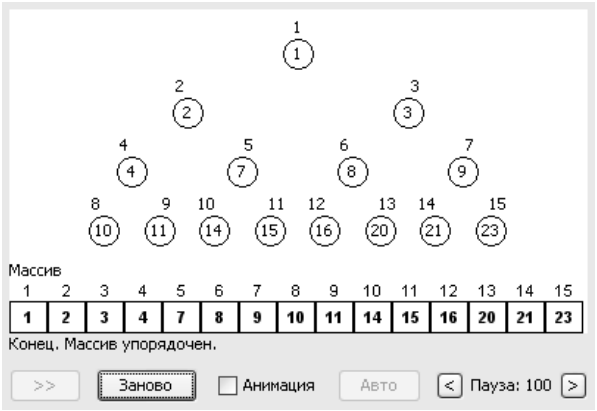

Рис. 79. Иллюстрация в состоянии 8

15. Обеспечение выбора в каждом анимационном состоянии статического или динамического изображения

Для отображения динамических иллюстраций, вершины, участвующие в обмене, отображаются на соединяющем их ребре (рис. 80).

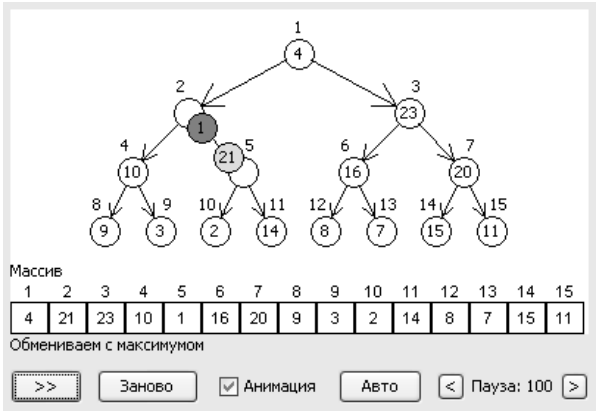

Рис. 80. Динамическая иллюстрация

16. Выбор архитектуры программы визуализатора

Для реализации пользовательского интерфейса сформирован еще один автомат, реализующий поведение интерфейса визуализатора. Схема взаимодействия этого автомата приведена на рис. 81.

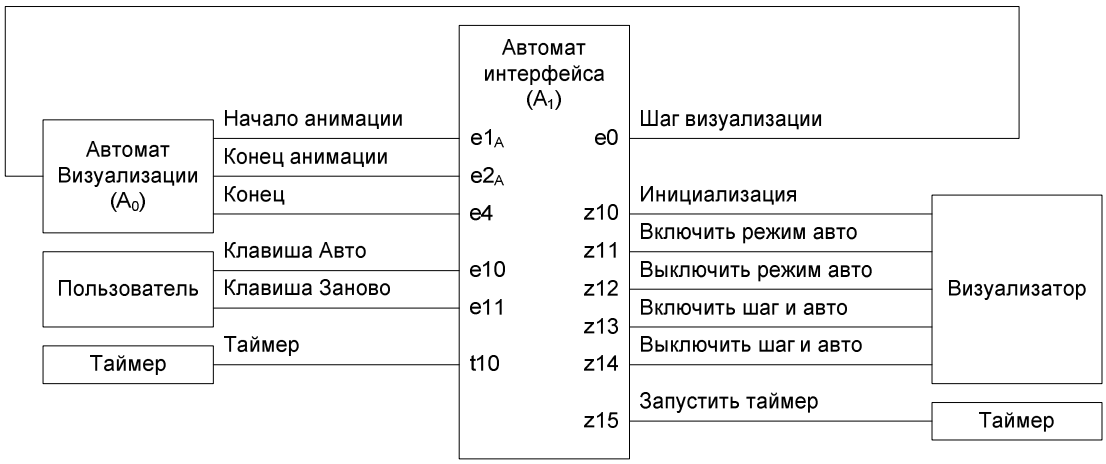

Рис. 81. Схема взаимодействия автомата интерфейса визуализатора

Как следует из этого рисунка, автомат взаимодействует с автоматом визуализации и управляет интерфейсом визуализатора. Таймер используется для реализации функции *Авто*. Диаграмма переходов интерфейсного автомата приведена на рис. 82.

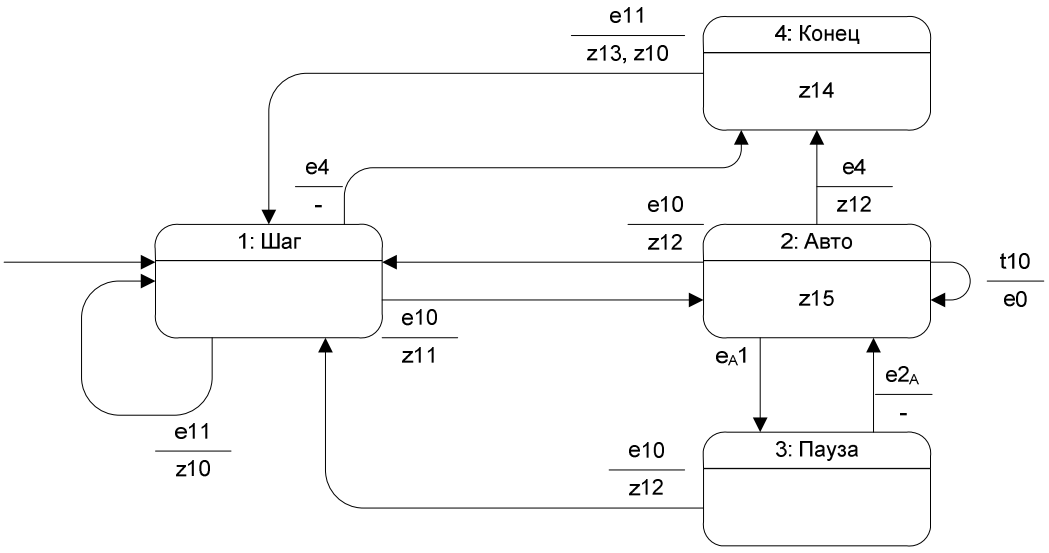

Рис. 82. Диаграмма переходов автомата интерфейса визуализатора

Схема взаимодействий преобразованного автомата визуализации приведена на рис. 65.

Схема взаимодействия всех компонент визуализатора представлена на рис. 83.

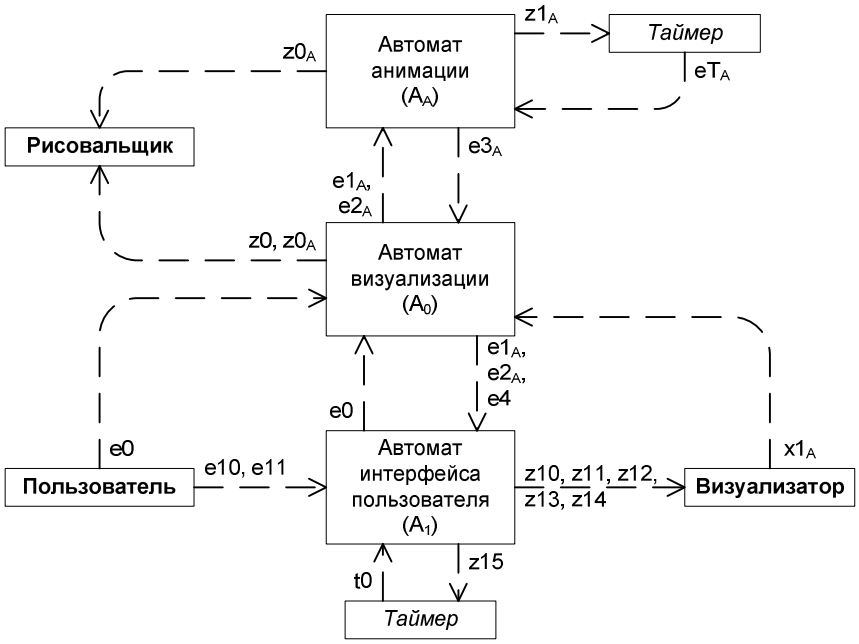

Рис. 83. Схема взаимодействия компонент визуализатора

17. Программная реализация визуализатора

Для построения визуализаторов предлагается использовать специальную разработанную автором библиотеку, предназначенную для реализации систем взаимодействующих автоматов. Основным принципом, заложенным в библиотеку, является прямое изоморфное преобразование схем автоматов Мили/Мура и схем их взаимодействия в специальные классы. При этом вся прикладная работа по запуску и взаимодействию автоматов осуществляется этой библиотекой. Достоинством библиотеки является возможность реализации, в том числе, и распределенной системы автоматов.

Исходный текст программы визуализатора приведен на сайте http://code.google.com/p/mk-thesis/

4.3. Построение визуализатора на примере алгоритма обхода двоичного дерева

Продемонстрируем метод на примере построения визуализатора алгоритма обхода двоичного дерева [2].

1. Постановка задачи обхода двоичного дерева

Задано двоичное дерево, состоящее из *n* вершин, каждая из которых имеет ссылки правого и левого потомка. Требуется реализовать алгоритм, который выводит по одному разу все вершины этого дерева [62].

2. Решение задачи

Приведем словесную формулировку решения этой задачи.

Для описания вершины дерева введем следующую структуру данных, описанную в листинге 10.

Листинг 10. Структура данных, описывающая вершину дерева

```
1       public class Node {
2      private int left;  // левое поддерево
3      private int right; // правое поддерево
4      private int index; // индекс в массиве
5      ...
6   }
```

Введем массив `visited[0..n-1]`, в котором будем отмечать пройденные вершины.

Для решения этой задачи необходимо создать вспомогательный стек вершин (*stack*). Далее, двигаясь от корня к листьям, предлагается совершать следующее действие в цикле:

- если в текущей вершине (*node*) левое поддерево не пройдено (*node.left* > 0 && !*visited*[*node.left*]), то записать ее в стек и двигаться к левому потомку;
- иначе, если правое поддерево не пройдено (*node.right* > 0 && !*visited*[*node.right*]), то записать текущую вершину в стек и двигаться к правому потомку;
- иначе, если в стеке находится хотя бы одно значение, то «снять» с вершины стека предыдущую вершину и двигаться к ней;
- иначе завершить алгоритм.

3. Выбор визуализируемых переменных

Предлагается визуализировать следующие переменные:

- *stack* – стек вершин от корня до текущей вершины;
- *visited* – массив флагов для идентификации пройденных вершин;
- *result* – результирующий массив;
- *node* – текущая вершина.

4. Выбор алгоритма визуализации

Предлагается визуализировать следующие особенности алгоритма для обеспечения возможности наилучшего разъяснения его действия:

- процесс занесения вершины в стек;
- текущая вершина;
- процесс перехода из предыдущей в следующую вершину;
- результат выполнения алгоритма;
- пройденные вершины (отмечать цветом).

5. Реализация алгоритма решения задачи

Приведем алгоритм решения задачи на псевдокоде [2]:

```
node ← nodes[0]
добавить node в результат
while true  do
    отметить node.left как пройденную
    if node.left ≥ 0 and node.left не пройдена
      then  добавить node в стек
            node ← node.left
            добавить node в результат
      else  if node.right ≥ 0
                        and node.right не пройдена
              then  добавить node в стек
                    node ← node.left
                    добавить node в результат
              else if стек не пуст
                        then node ← с вершины стека
                        else break
return результат
```

6. Реализация алгоритма на языке *Java*

Перепишем программу, записанную на псевдокоде, с помощью языка *Java*.

```
1       /**
2        * @param nodes массив вершин дерева
3        * @return список всех вершин, выведенных в порядке обхода
4        */
5       private static List traverse(Node[] nodes) {
6           // Содержит результат
7           List result = new ArrayList();
8           // Стек для обеспечения возврата
9           LinkedList stack = new LinkedList();
10          // Хранят флаги того, что вершина пройдена
11          boolean[] visited = new boolean[nodes.length];
12          // Заполняем исходными значениями
13          Arrays.fill(visited, false);
14          // Выбираем корневую вершину
15          Node node = nodes[0];
16          result.add(node);
17          while (true) {
18              // Отмечаем текущую вершину, как пройденную
19              visited[node.index()] = true;
20              if (node.left() >= 0 && !visited[node.left()]) {
21                  // Переход к левому поддереву
22                  stack.addLast(node);
23                  node = nodes[node.left()];
24                  result.add(node);
25              } else if (node.right() >= 0 && !visited[node.right()]) {
26                  // Переход к правому поддереву
27                  stack.addLast(node);
28                  node = nodes[node.right()];
29                  result.add(node);
30              } else {
31                  // Возврат
32                  if (stack.isEmpty()) {
33                      break;
34                  }
35                  node = (Node)stack.removeLast();
36              }
37          }
38          return result;
39      }
```

В этой программе операции ввода/вывода не приведены, так как их будет выполнять визуализатор.

7. Построение схемы алгоритма по программе

Построим по тексту программы схему алгоритма (рис. 84).

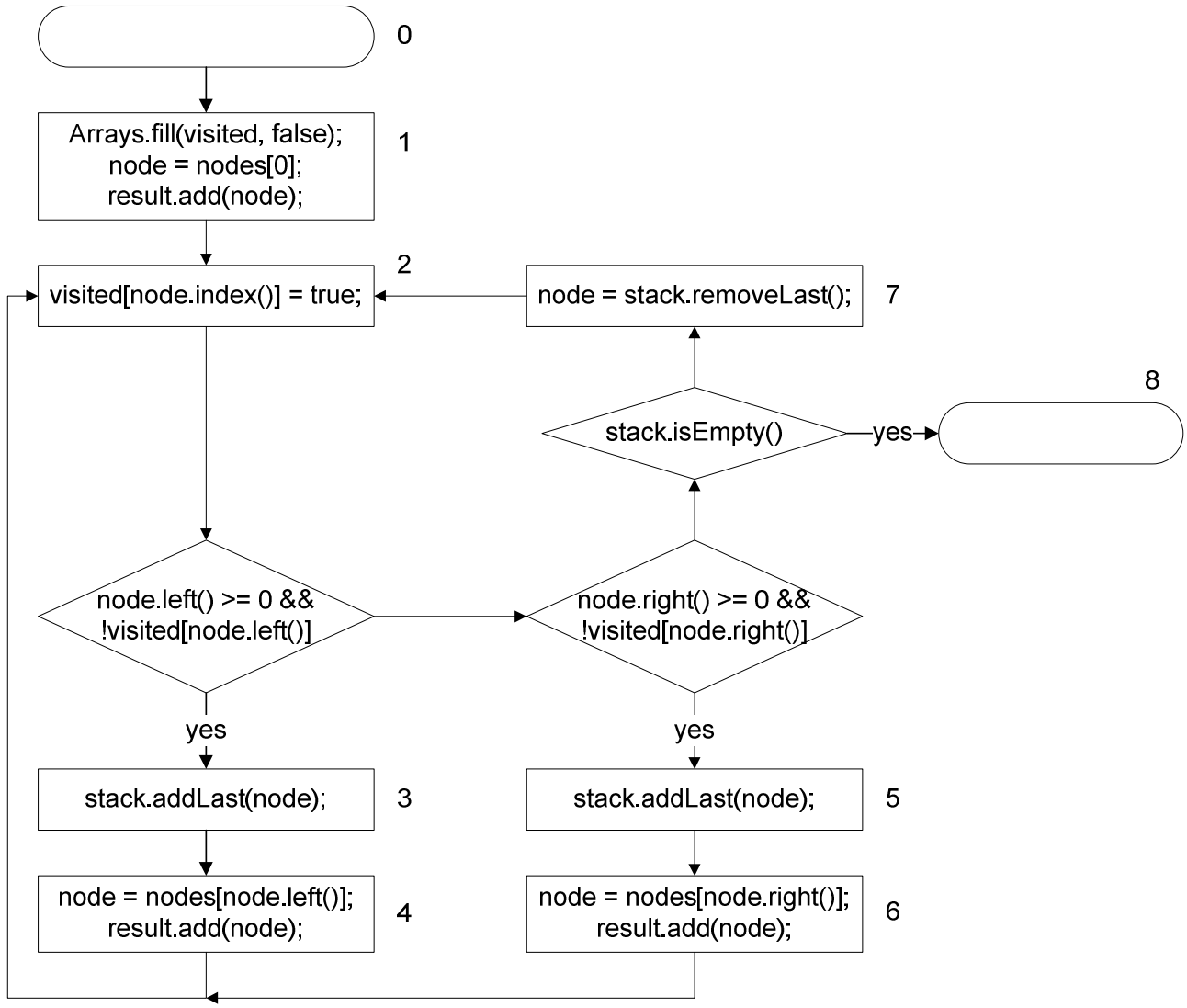

Рис. 84. Схема алгоритма двоичного обхода дерева

8. Преобразование схемы алгоритма в автомат Мура

Следуя методу, приведенному в работе [58], построим автомат Мура, соответствующий приведенной выше схеме алгоритма (рис. 85).

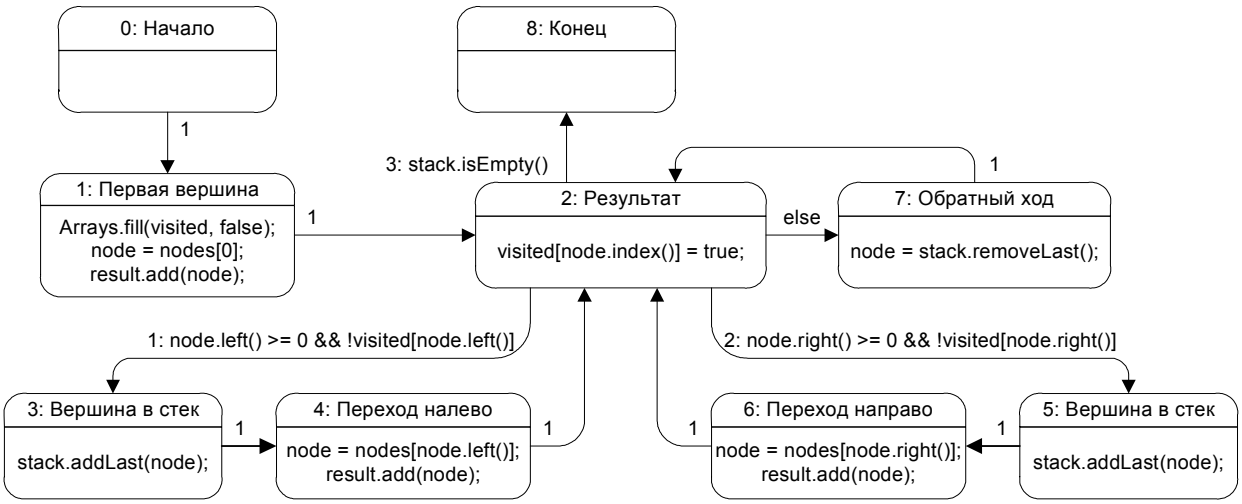

Рис. 85. Автомат Мура, реализующий обход двоичного дерева

Исходя из схемы алгоритма на рис. 84, выделим девять состояний. В состоянии 1 выполняется инициализация алгоритма. В состоянии 2 текущая вершина отмечается как пройденная. В состояниях 3 и 5 текущая вершина помещается в стек, а в состояниях 4 и 6 выполняется переход. Состояние 7 отвечает за обратный ход. Состояние 8 является конечным.

9. Преобразование автомата, реализующего алгоритм в автомат визуализации

В соответствии с методом, изображенным на рис. 63, автомат, изображенный на рис. 85, преобразуется в автомат, представленный на рис. 86.

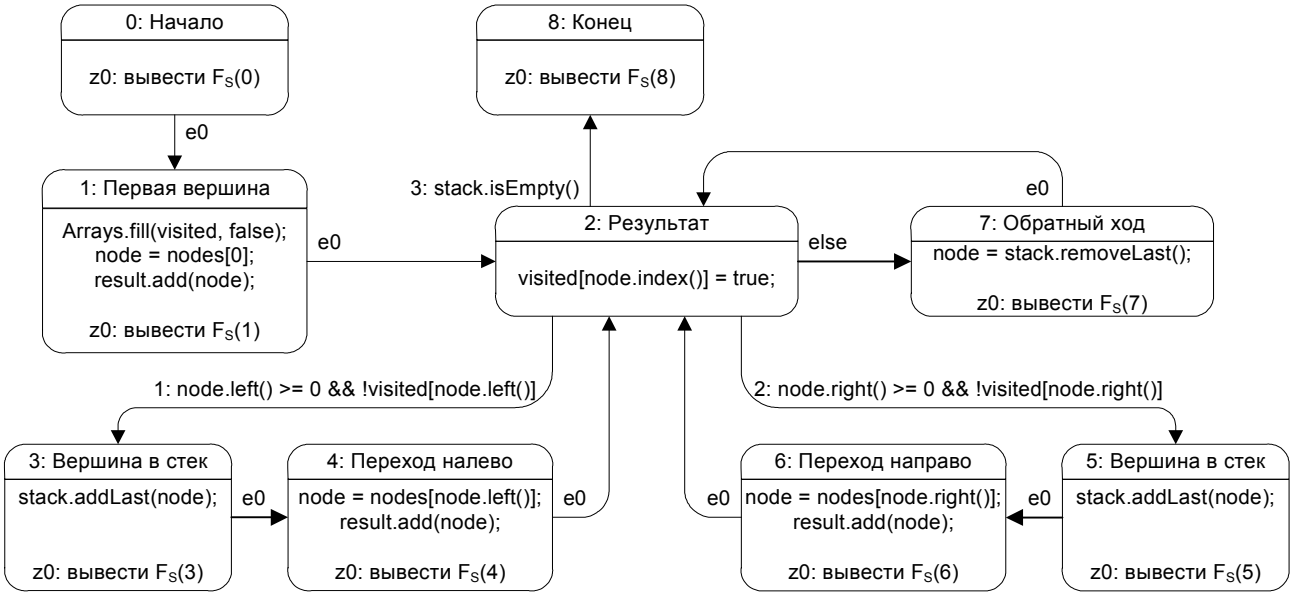

Рис. 86. Автомат визуализации

10. Выбор состояний, в которых будет выполняться анимация

В алгоритме обхода дерева наиболее важным для отображения является процесс передвижения по вершинам, поэтому анимационными являются состояния, в которых изменяется номер текущей вершины. Такими состояниями являются состояния 4, 6, 7.

11. Замена каждого из анимационных состояний тремя состояниями

В соответствии с методом, изложенным выше (рис. 64), заменим состояния 4, 6, 7 тройками состояний, в первом из которых запоминается номер предыдущей вершины, во втором формируется событие начала анимации, а в третьем завершается шаг визуализатора и создается конечная иллюстрация в состоянии (рис. 87). При этом событие окончания анимации формируется на переходе между анимационным и конечным состояниями шага визуализации.

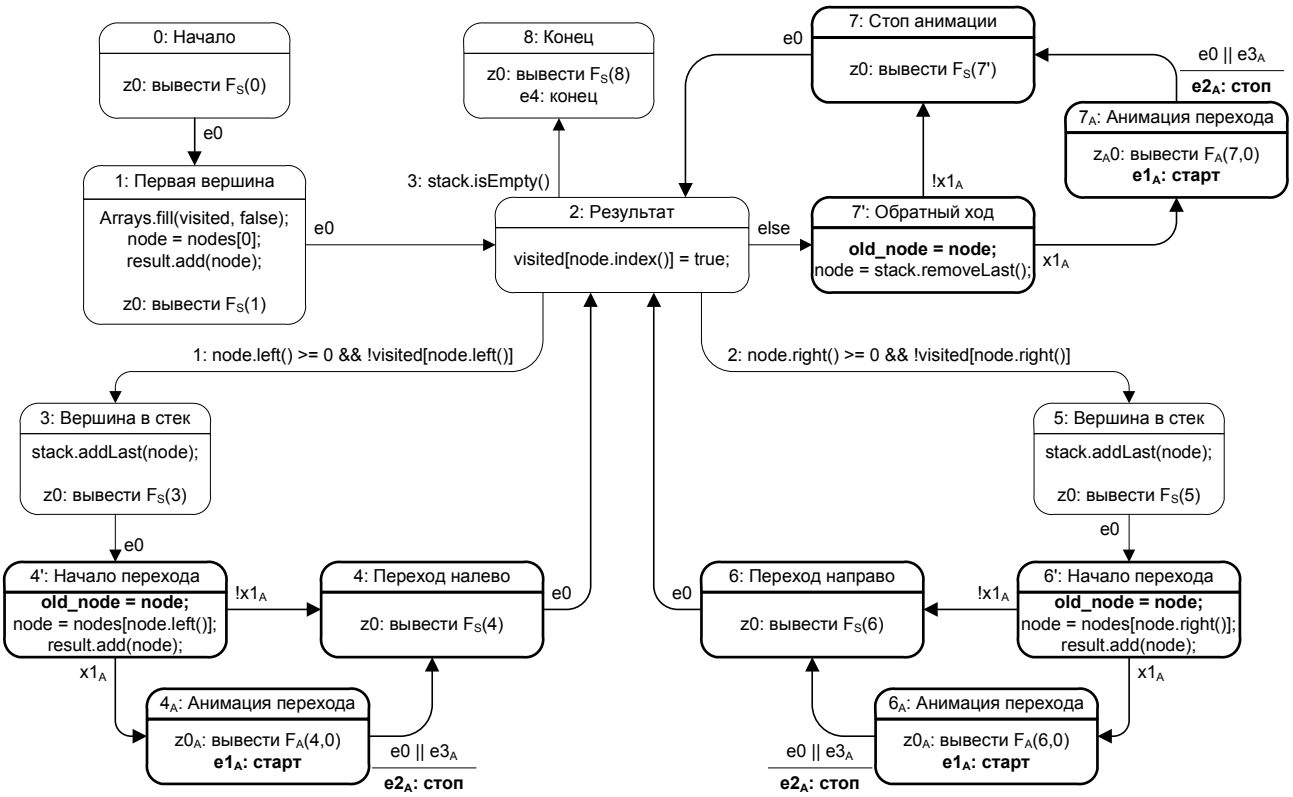

Рис. 87. Преобразованный автомат визуализации

12. Использование автомата анимации и обеспечение его взаимодействия с преобразованным автоматом визуализации

Как отмечалось выше, для анимации используется стандартный автомат, приведенный на рис. 66. Для обеспечения взаимодействия между автоматом визуализации и автоматом анимации вводятся связи, представленные на рис. 67.

13. Выбор интерфейса визуализатора

В верхней части визуализатора (рис. 88) приводится следующая информация:

1. Дерево, изображенное графически. Серым цветом отмечены пройденные вершины, а не пройденные – белым цветом. Текущая вершина отмечена темно серым цветом;
2. В правой части иллюстрации изображен стек, а в нижней – порядок обхода дерева (массив пройденных вершин).

Под массивом пройденных вершин динамически отображаются текстовые комментарии, соответствующие текущим состояниям автомата визуализации. Например, на рис. 88 изображен комментарий «Двигаемся направо», соответствующий состоянию 6. Под комментарием расположены элементы управления. Следует отметить, что по сравнению с обычным визуализатором появляется флаг управления анимацией:

1. «>>» – сделать шаг алгоритма.
2. «Заново» – начать алгоритм сначала.
3. «Анимация» – отображать или не отображать анимацию.
4. «Авто» – перейти в автоматический режим.
5. «<», «>» – изменить паузу между автоматическими шагами алгоритма.

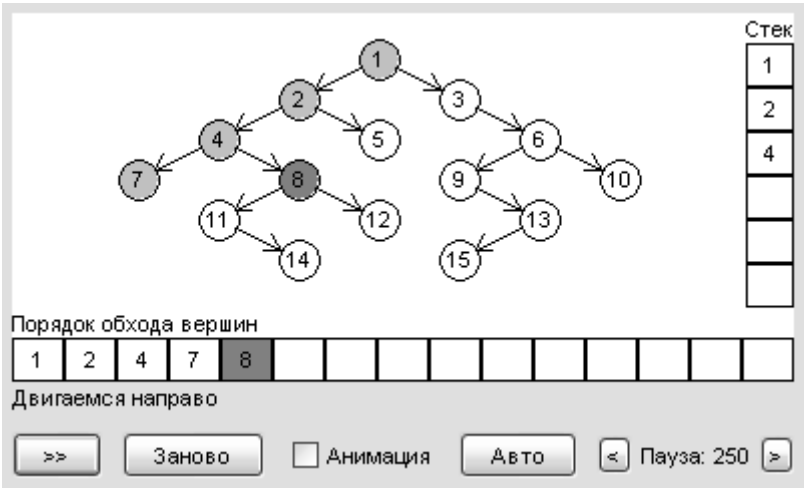

Рис. 88. Интерфейс визуализатора

14. Сопоставление иллюстраций и комментариев с состояниями автомата

Для наглядности визуализации алгоритма предлагается акцентировать внимание на следующих элементах.

В состояниях 4 и 6 (рис. 88) статическая иллюстрация отображает:

1. Текущую вершину.
2. Пройденные вершины.
3. Последнюю вершину в результирующем массиве.

В состояниях 3 и 5 (рис. 89) отмечается вершина, добавленная в стек.

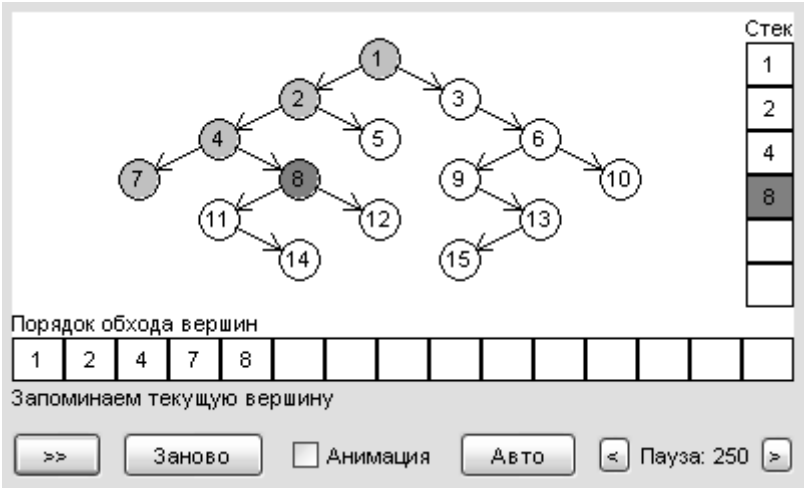

Рис. 89. Иллюстрация в состояниях 3 и 5

В состоянии 7 (рис. 90) отображается обратный ход. При этом отмечается только текущая вершина, поскольку данные в стек и массив пройденных вершин не добавляются.

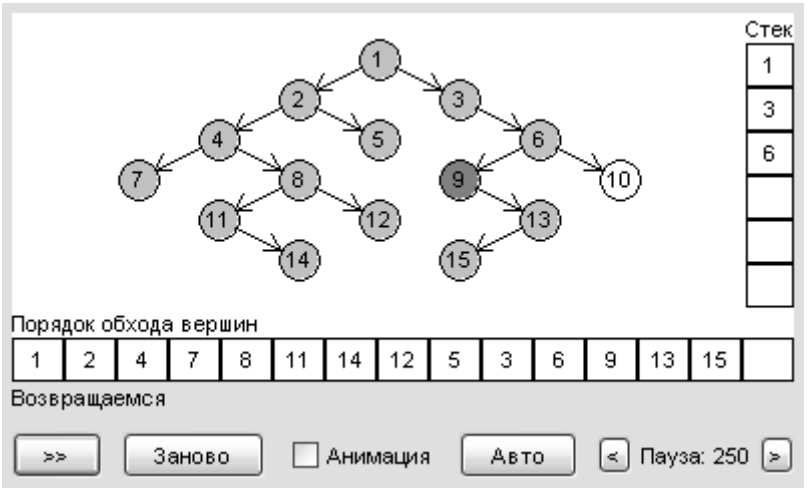

Рис. 90. Иллюстрация в состоянии 7

15. Обеспечение выбора в каждом анимационном состоянии статического либо динамического изображения

Для отображения динамических иллюстраций используются статические иллюстрации с наложенным изображением движущегося указателя (рис. 91).

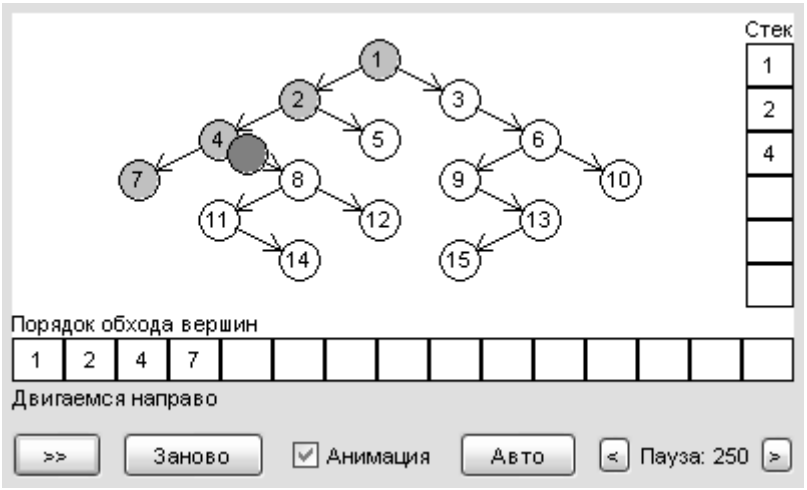

Рис. 91. Динамическая иллюстрация

16. Выбор архитектуры программы визуализатора

Для реализации пользовательского интерфейса сформирован еще один автомат, рассмотренный в разд. 4.2, реализующий поведение интерфейса визуализатора. Схема взаимодействия этого автомата приведена на рис. 81.

Диаграмма переходов интерфейсного автомата приведена там же на рис. 82. При этом схема взаимодействий преобразованного автомата

визуализации изображена на рис. 65. Схема взаимодействия всех компонент визуализатора представлена на рис. 83.

17. Программная реализация визуализатора

Исходный текст программы визуализатора приведен на сайте http://code.google.com/p/mk-thesis/

Выводы по главе 4

1. В главе предложен **формализованный** подход к реализации визуализаторов алгоритмов с анимацией, улучшающий восприятие алгоритмов обучающимися, что особенно важно при дистанционном обучении. Он расширяет метод построения визуализаторов алгоритмов на основе автоматного подхода, предложенный в главе 2.
2. Также в этой главе приведены два примера различной природы. Несмотря на различие визуализируемых алгоритмов, применение метода обеспечения анимации привело к сходным результатам, что подтверждает факт универсальности предложенного подхода.

ГЛАВА 5. ВНЕДРЕНИЕ РЕЗУЛЬТАТОВ РАБОТЫ В УЧЕБНЫЙ ПРОЦЕСС

С 1997 по 2004 г. автор проводил курсовые работы по дисциплине «Дискретная математика» для студентов первого курса кафедры «Компьютерные технологии» СПбГУ ИТМО. Задачей, которая ставилась перед студентами, являлась построение визуализаторов алгоритмов. При этом студенты строили визуализаторы алгоритмов на основе эвристического подхода без использования автоматов. Часть списка курсовых работ, выполненных под руководством автора, приведена в табл. 6.

Таблица 6. Курсовые работы, выполненные под руководством автора

№	Студент	Алгоритм
1.	Золотухин Ю.	*LZ*-сжатие
2.	Добровольский В.	Перебор перестановок
3.	Крылов Р.	Обход дерева
4.	Михалев Е.	Генератор перестановок
5.	Прокушкин И.	Алгоритм Кнута-Морриса-Пратта
6.	Филоненко Ф.	Сортировки массива
7.	Порох Ю.	Поиск выпуклой оболочки
8.	Аничкин И.	Венгерский метод
9.	Альхов С.	Хэш-код
10.	Белов Д.	Сортировка фон Неймана
11.	Бородин Т.	Алгоритм Кнута-Морриса-Пратта
12.	Наумов Л.	Алгоритмы поиска подстрок

Как отмечалось выше (разд. 1.4), традиционный эвристический способ построения визуализаторов без использования автоматов, применяемый студентами, был несистематическим и процесс выполнения каждого визуализатора превращался в многочасовые многодневные совместные усилия студента с преподавателем. Среднее время преподавателя, затрачиваемое на визуализатор, равнялось приблизительно 10 часам, а время, затрачиваемое студентом, при этом превышало 100 часов.

Начиная с 2002 г. при разработке визуализаторов в рамках курсовых, бакалаврских и магистерских работ автором внедрялся автоматный подход к построению визуализаторов и, в частности, метод, изложенный в рамках данной диссертации. При этом в различное время студенты использовали разные варианты автоматного подхода при построении визуализаторов в своих работах.

5.1. Применение эвристического автоматного подхода

В курсовой работе студента Георгия Удова в 2004 г. [14] автор выступал в качестве консультанта. Целью данной работы было сравнение традиционного и эвристического автоматного подходов.

В своей работе студент Г. Удов рассматривал построение визуализатора алгоритма пирамидальной сортировки [63], который является одним из фундаментальных методов сортировки набора чисел. Выбор студента остановился на этом алгоритме, поскольку он является одним из лучших алгоритмов сортировки массива по производительности, так как требует O(1) дополнительной памяти, а его трудоемкость в худшем случае составляет O(n log(n)), что является лучшей для общего случая сортировок. При построении этого алгоритма используется нетипичный для сортировок подход – применяется структура данных, называемая пирамидой [2].

Как отмечалось выше, в работе выполнены две реализации алгоритма пирамидальной сортировки набора чисел – классическая [8, 78] и с использованием эвристического автоматного подхода, и было произведено их сравнение.

Следуя рекомендациям автора диссертации, в качестве технологии выбрана технология *Java Applets*. При построении двух реализаций визуализатора Г. Удов придерживался эвристического подхода без и использованием автоматов. При этом в качестве базового подхода к реализации визуализаторов использовался подход, названный в первой главе как «Стек состояний». Суть подхода состоит в том, что алгоритм проходит от начала до

конца и запоминаются контрольные точки, которые требуется визуализировать. Далее во время работы визуализатора происходит последовательная смена иллюстраций, соответствующих визуализируемым контрольным точкам.

При сравнении двух подходов автор курсовой работы реализовал функцию, формирующую стек состояний двумя способами.

1. Традиционный способ, основанный на алгоритме, изложенном в книге [3].
2. Способ, основанный на *SWITCH*-технологии [59], который по своей сути и является эвристическим автоматным подходом к построению визуализаторов алгоритмов, описанным в разд. 2.1 настоящей работы.

При использовании автоматного подхода студент Г. Удов разработал неформализованную автоматную реализацию алгоритма пирамидальной сортировки. При этом он применил следующую реализацию алгоритма сортировки (листинг 11) [].

Листинг 11. Алгоритм пирамидальной сортировки, использованный студентом

```
1  void heapSort() {
2    l = (number / 2);
3    r = number - 1;
4    while(l > 0) sift(--l, r);
5    while(r > 0) {
6      swap(0, r--);
7      sift(l, r);
8    }
9  }
```

Поскольку основную часть алгоритма реализует два цикла, которые с его точки зрения являются неважными в процессе визуализации, то он принял решение, что автоматной реализации требует именно реализация метода *sift()*, выполняющего «просеивание» элементов. Тем более именно в этом методе и состоит основная сложность при визуализации – выделение состояний. При реализации функции *sift()* был получен следующий автомат (рис. 92).

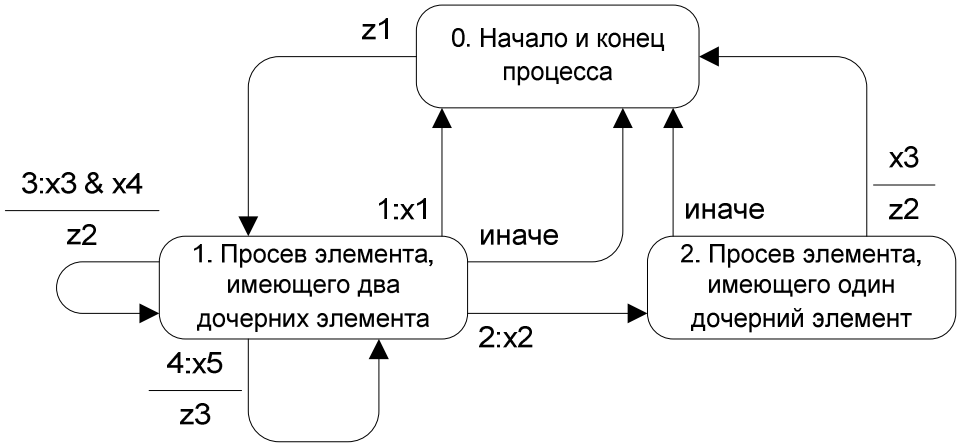

Рис. 92. Автомат, реализующий метод *sift()*

При этом использовалась следующая схема связей автомата:

- *x*1 – просеиваемый элемент не имеет дочерних;
- *x*2 – просеиваемый элемент имеет только один дочерний элемент;
- *x*3 – просеиваемый элемент меньше своего левого ребенка;
- *x*4 – левый ребенок просеиваемого элемента не меньше первого;
- *x*5 – просеиваемый элемент меньше своего правого ребенка;
- *z*1 – установить значением индекса текущего просеиваемого элемента индекс массива, с которого начинается процесс просева;
- *z*2 – поменять местами значения и индексы текущего просеиваемого элемента и его левого ребенка;
- *z*3 – поменять местами значения и индексы текущего просеиваемого элемента и его левого ребенка.

После построения такого решения состояния системы выявляются автоматически: состояния 1 и 2 автомата на рис. 92 являются визуализируемыми. Таким образом, визуализатор алгоритма построен на основе автоматного эвристического подхода.

После окончания работы студент Г. Удов делает следующие выводы:

1. «*В работе приведены две реализации алгоритма пирамидальной сортировки – классическая и автоматная. Автоматная реализация расширяет использование конечных автоматов в программировании. Их применение делает естественным процесс визуализации, так как,*

не вводя понятие состояния, в общем случае не понятно, в какой момент необходимо выполнять шаг визуализации.»

2. *«В силу того, что в функции sift состояния* ***выделились естественно****, то для нее автоматный подход может применяться не только при построении визуализатора, но и непосредственно при реализации*» [14].

5.2. Применение автоматного подхода

Автор настоящей работы, как уже отмечалось, занимается визуализаторами алгоритмов с 1997 г. При этом до 2001 г. визуализаторы строились традиционным, эвристическим способом. В 2001 г. на лекционных занятиях профессора А.А. Шалыто, автор настоящей работы ознакомился со *SWITCH*-технологией [59] и автоматным программированием [44]. В рамках семинарских занятий профессор предложил студентам апробировать знания, полученные на лекциях на практике.

5.2.1. Применение автоматного подхода для алгоритма Дейкстры

Для апробации *SWITCH*-технологии и, в частности, метода преобразования итеративных программ в автоматные, автор данной работы выбрал один из классических алгоритмов теории графов – алгоритм Дейкстры для поиска кротчайшего расстояния в графе от одной вершины до остальных [5]. Задача, решаемая этим алгоритмом, формулируется следующим образом.

Дан взвешенный ориентированный граф G(V,E) без петель и дуг отрицательного веса. Найти кратчайшие пути от некоторой вершины a графа G до всех остальных вершин этого графа.

Сформулируем алгоритм Дейкстры. Введем следующие обозначения.

- n – число вершин в графе;
- $r[i][j]$ – массив длин ребер от вершины i до вершины j. Отрицательное значение в ячейке обозначает отсутствие ребра.

- *len*[*i*] – искомый массив (минимальное расстояние от вершины *a* до вершины *i*). Изначально заполнен бесконечностями.

Для решения задачи введем дополнительные переменные.

- *used*[*i*] – массив флагов. Изначально заполнен *false*;
- *k* – текущая ближайшая к *a* вершина.

Алгоритм на псевдокоде формулируется следующим образом:

```
1    выбрать в качестве текущей вершины k вершину a
2    заполнить len[a] = 0
3    повторить n-1 раз
4      отметить вершину k как пройденную used[k] = true
5      для всех не пройденных вершин i проверить
6        если расстояние len[i] > len[k] + r[k][i]
7          то подменить len[i] = len[k] + r[k][i]
8      найти не пройденную вершину с минимальным
9        расстоянием len[i] и запомнить ее номер в k
```

Приведенный алгоритм реализуется следующим образом на языке *Java*:

```
1   k = a;
2   len[a] = 0;
3   for (int j = 0; j < n-1; j++) {
4     used[k] = true;
5     for (int i = 0; i < n; i++)
6       if (!used[i] && r[k][i] >= 0 && len[i] > len[k] + r[k][i])
7         len[i] = len[k] + r[k][i];
8     for (int i = 0; i < n; i++)
9       if (!used[i] && (used[k] || len[i] < len[k])) k = i;
10  }
```

Как видно из реализации алгоритма, эта, достаточно сложная математическая задача, имеет очень короткое традиционное решение. В рамках использования метода перевода итеративных программ в автоматные [58, 79] автор данной работы произвел стандартные шаги. Первый шаг при преобразовании итеративной программы в эквивалентный автомат, согласно указанному методу, состоит в построении схемы алгоритма и выделении состояний.

Первый способ, который был апробирован, – это преобразование схемы алгоритма в автомат Мура. При таком преобразовании в состояния

преобразуются последовательности операторных вершин на схеме алгоритма (рис. 93). На этом рисунке состояния отмечены цифрами.

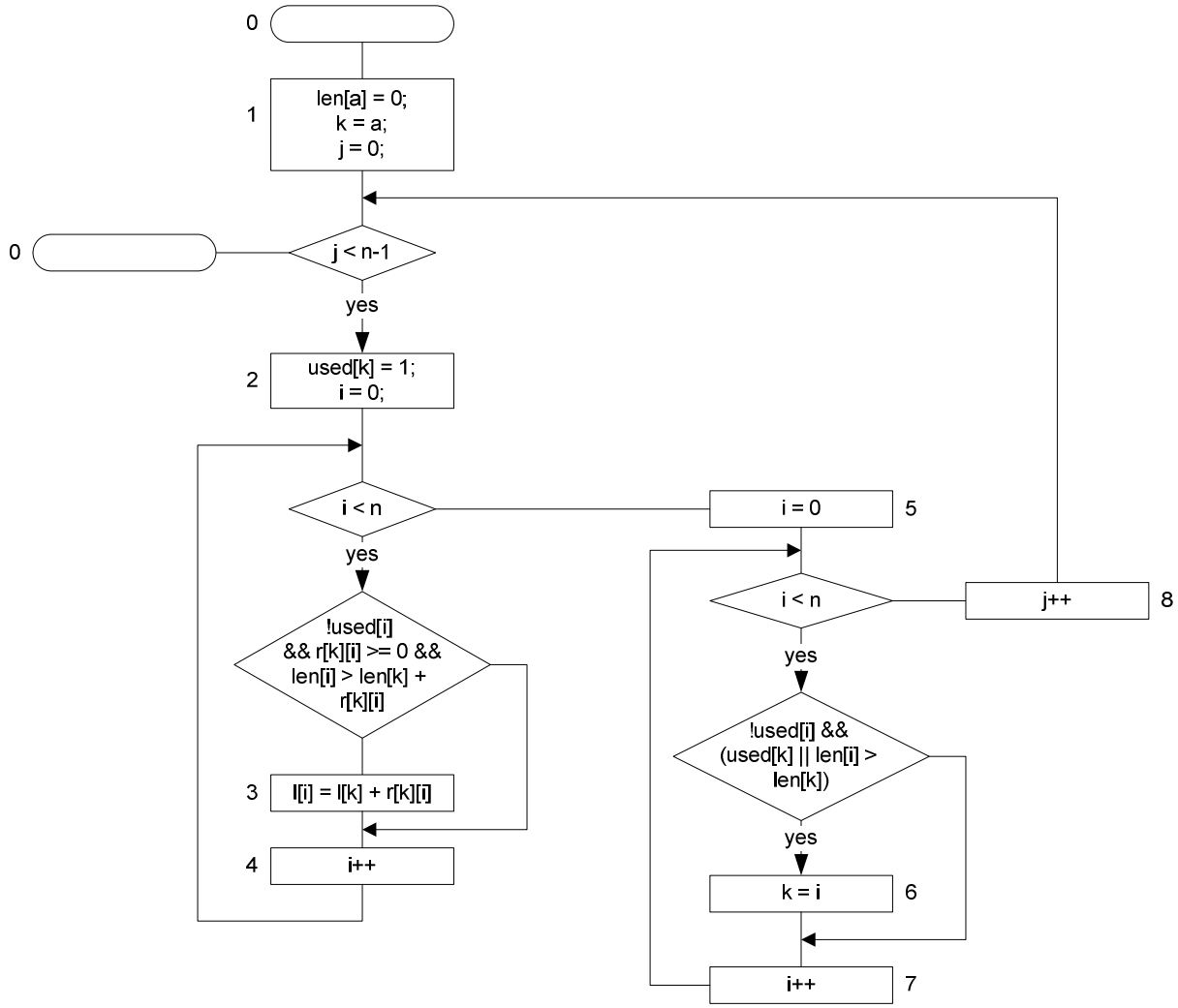

Рис. 93. Схема алгоритма Дейкстры с отмеченными состояниями автомата Мура

Проходя все возможные пути между выделенными состояниями, строится автомат Мура, приведенный на рис. 94.

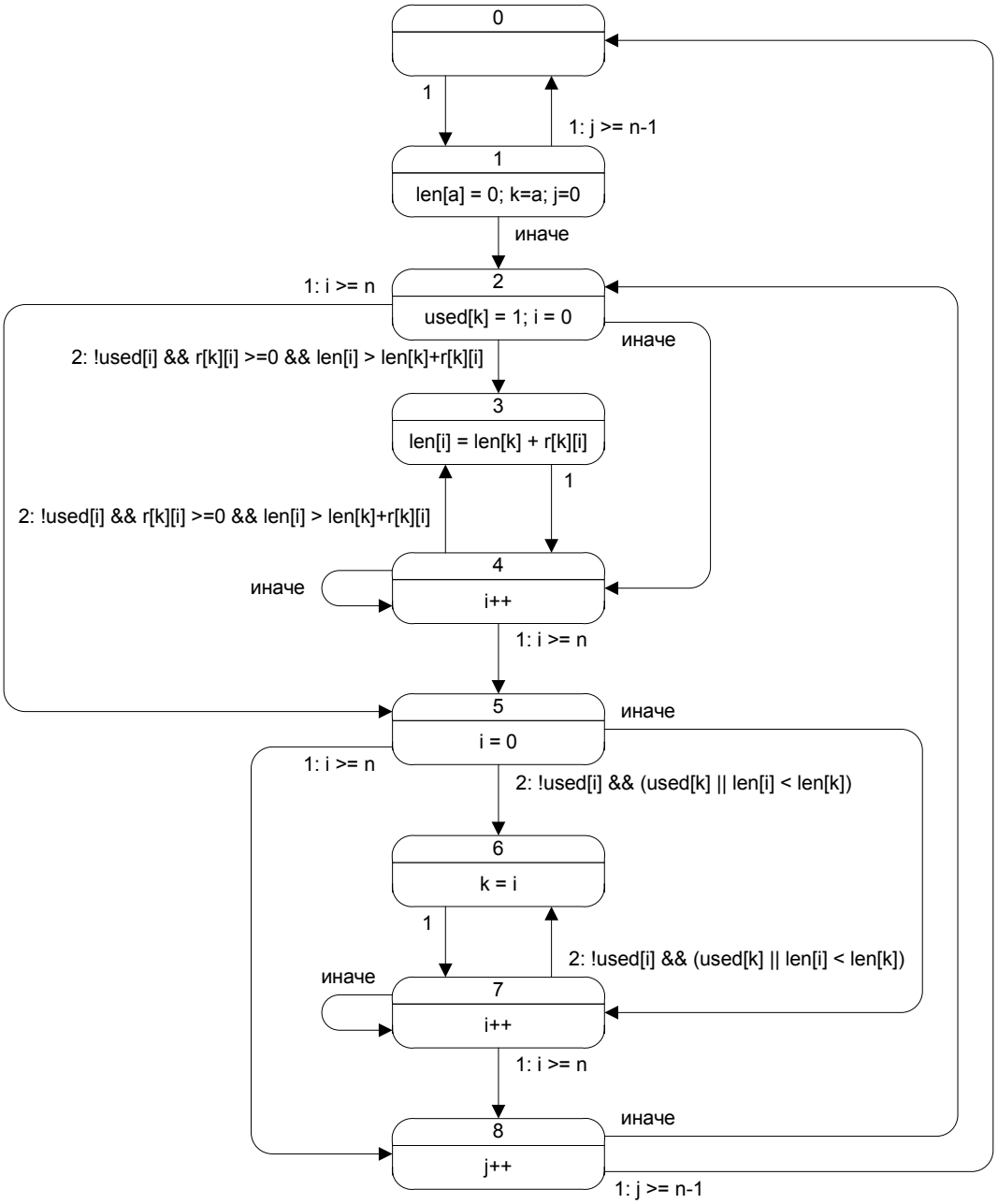

Рис. 94. Автомат Мура реализующий алгоритм Дейкстры

С первого взгляда стало ясно, что сложность автомата существенно выше, чем сложность реализации алгоритма, написанного традиционным способом на языке *Java*.

Однако, существует второй вариант преобразования итеративных программ в автоматные с использованием автоматов Мили. При этом состояния назначаются в точках, следующих за последовательностями операторных вершин. В рассматриваемом случае автомат Мили будет иметь четыре состояния (рис. 95, рис. 96).

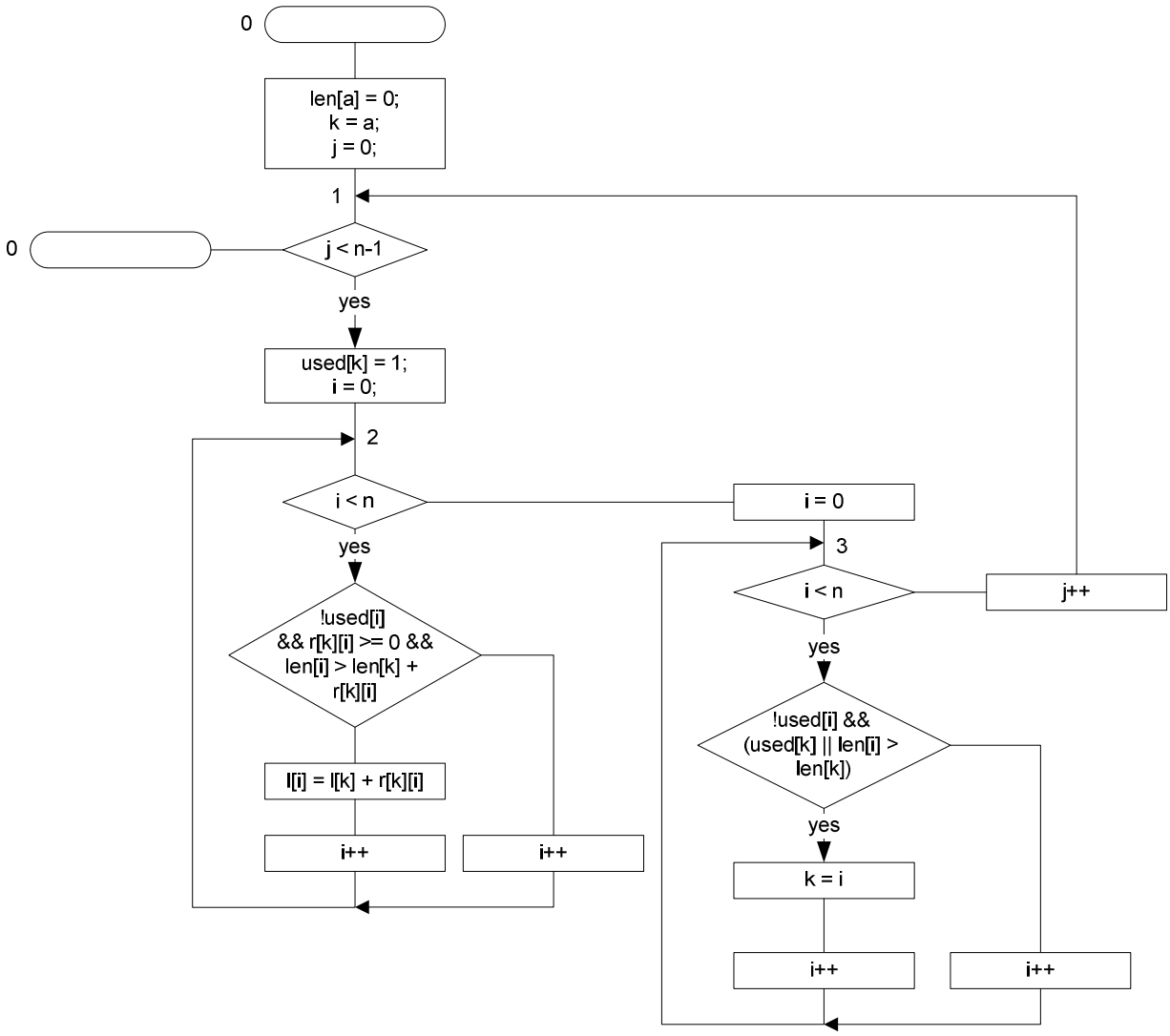

Рис. 95. Схема алгоритма Дейкстры с отмеченными состояниями автомата Мили

Для уменьшения числа состояний в автомате Мили операторные вершины, в которых осуществлялся инкремент индексных переменных, были раздвоены (рис. 96).

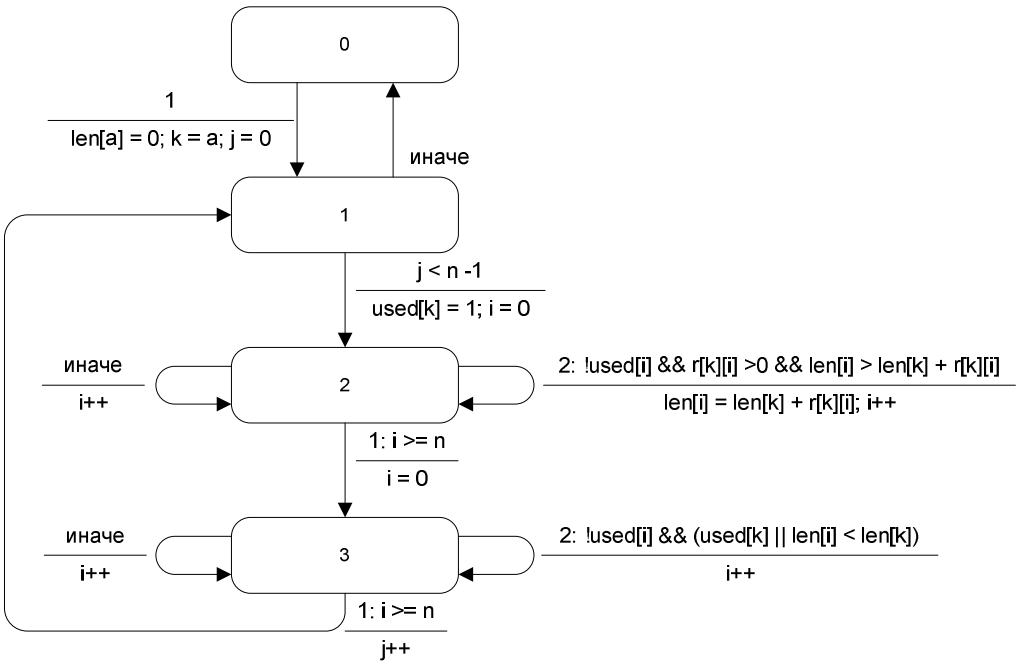

Рис. 96. Автомат Мили эквивалентный алгоритму Дейкстры

Таким образом, попытки использовать формальное преобразование итеративных алгоритмов в автоматные с точки зрения не только не дало никаких видимых преимуществ по сравнению с традиционным подходом, но и приводит к более громоздким реализациям.

На вопрос профессора А.А. Шалыто: «Есть ли смысл использовать автоматное программирование для реализации вычислительных алгоритмов?» – автором данной работы ответил: «Нет, это в большинстве случаев нецелесообразно, однако такой подход может быть крайне полезен при построении визуализаторов алгоритмов». С этого момента начал разрабатываться и применяться автоматный подход при реализации визуализаторов алгоритмов [68, 69, 71, 81, 82, 84, 86].

5.2.2. Применение автоматного подхода студентами

Начиная с 2002 г. студенты кафедры «Компьютерные технологии» используют формализованный автоматный подход при разработке визуализаторов в рамках курсовых работ по курсу «Дискретная математика».

В общей сложности с 2002 по 2010 г. на основе метода формализованного построения визуализаторов было спроектировано и

построено более 100 визуализаторов алгоритмов, многие из которых опубликованы на сайтах [12] и [64], что подтверждено актом внедрения.

Пример одной из работ, в которой непосредственно использовался предлагаемый метод, приведен в Приложении, в котором в 2004 г. студентами В.А. Добровольским и А.В. Степуком был разработан визуализатор «Сортировка подсчетом» [65]. На создании этого визуализатора студенты потратили семь часов, а преподаватель – всего один час.

5.3. Использование в Интернет-школе программирования

Проект Интернет-школа программирования [26] предложен, реализован и действует на кафедре «Компьютерные технологии» с 1999 г. [73, 83, 85, 87–90]. С начала преподавания в Интернет-школе уроки, опубликованные на сайте, снабжались визуализаторами. Если до конца 2001 г. визуализаторы разрабатывались традиционным методом, то, начиная с 2002 г. в процессе построения визуализаторов, начал использоваться метод, предложенный в настоящей работе, что существенным образом отразилось на их качестве и скорости построения визуализаторов. При этом сначала использовался ручной метод, а в дальнейшем визуализаторы стали создаваться на базе системы визуализации *Vizi*, которая основана на совместной статье автора с Г.А. Корнеевым [74].

Все визуализаторы, рассмотренные в рамках данной диссертации, в настоящее время опубликованы на сайте Интернет-школы программирования.

5.4. Дальнейшее развитие предлагаемого подхода

Подход, предложенный автором, может использоваться и для более широкой области. Так, например, под руководством автора в 2006 г. была выполнена работа А. Корниенко по теме «Автоматный подход к реализации элементов графического пользовательского интерфейса» [66]. В ней весьма успешно было предложено использовать автоматы для визуализации пользовательского интерфейса. При этом отметим, что идея применение автоматов для визуализации пользовательского интерфейса является, казалось

бы, не особо новой, но это не так. Так, только в 2007 г. ведущий программист компании *IBM* опубликовал три (!) [15 – 17] статьи о применении автоматов для реализации всплывающих подсказок. Отсутствие формализованного метода в его работе привело к ошибке, что отмечено в работе О. Коломейцевой [18].

5.5. Сравнение трудозатрат на создание визуализаторов различными методами

На основании опыта преподавания на кафедре «Компьютерные технологии» был собран статистический материал по трудоемкости создания визуализаторов алгоритмов, который обобщен в табл. 7.

Таблица 7. Сравнение трудозатрат на разработку визуализатора в часах

Подход	**Затраты студента**		**Затраты преподавателя**
	простые алгоритмы	сложные алгоритмы	
Традиционный	80–120	120–180	8–10
Автоматный эвристический	10–20	40–80	2–3
Формализованный автоматный	2–5	20–40	менее одного часа
Система *Vizi*	5–10	10–20	менее одного часа

Таким образом, предлагаемый подход для простых алгоритмов эффективнее даже автоматизированного подхода, так как время, требуемое для освоения системы *Vizi*, больше времени, требуемого для реализации визуализатора. Такие результаты соответствуют результатам исследований в области кривой обучения [67].

Выводы по главе 5

В главе 5 описаны результаты внедрения результатов работы в учебный процесс.

1. Использование традиционного подхода к построению визуализаторов алгоритмов без использования автоматов требовало значительных трудозатрат, при этом качество визуализаторов было достаточно низким.
2. Использование эвристического автоматного подхода существенно сократило и упростило процесс написания визуализаторов алгоритмов в тех случаях, когда он был применим.
3. Применение формализованного подхода к построению визуализаторов не только снизило затраты на создание как со стороны студентов, так и со стороны преподавателя во много раз, но и существенно улучшило качество исполняемых визуализаторов.
4. Визуализаторы алгоритмов являются важным инструментальным средством в проекте дистанционного обучения «Интернет-школа программирования».

ЗАКЛЮЧЕНИЕ

Анализ опыта разработки и применения визуализаторов алгоритмов в учебном процессе показал, что их разработка велась на основе эвристических подходов, что существенно отражалось на трудозатратах, как их разработчиков, так и преподавателей.

В ходе проведенных исследований были разработаны методы формального построения визуализаторов алгоритмов дискретной математики. Их применение обеспечило индивидуализацию практических и самостоятельных занятий студентов в СПбГУ ИТМО с использованием эффективных средств автоматизации образовательного процесса в интерактивной форме.

В диссертации получены следующие результаты.

1. В ходе проведенного анализа установлено, что для построения визуализаторов алгоритмов дискретной математики эффективной основой является автоматной подход, который естественным образом позволяет выделять и визуализировать состояния действий алгоритма.

2. Разработан метод формализованного построения визуализаторов алгоритмов дискретной математики на основе конечных автоматов, который имеет не только все достоинства автоматного подхода, но и дает четкую последовательность действий для перехода от визуализируемого алгоритма к его визуализатору.

3. Разработан метод построения визуализаторов алгоритмов дискретной математики на основе системы взаимодействующих конечных автоматов, который позволил проводить визуализацию алгоритмов как пошагово, так и поэтапно.

4. Разработан метод обеспечения простой анимации в визуализаторах на основе конечных автоматов. Использование этого метода в визуализаторах позволило анимировать динамические иллюстрации.

5. При внедрении предложенного *автоматного эвристического подхода* в процесс построение визуализаторов, затраты разработчика снизились в шесть–восемь раз по сравнению с традиционным эвристическим подходом, при этом время, затраченное преподавателем снизилось в три–четыре раза.

6. При внедрении *автоматного формализованного метода* к построению визуализаторов алгоритмов трудозатраты на разработку одного визуализатора снизились до 30 раз по сравнению с традиционным подходом, а время, затраченное преподавателем, снизилось в 10 раз.

7. Предложенные методы успешно используются с 2002 г. в учебном процессе на кафедре КТ СПбГУ ИТМО. За разработку и внедрение методов построения визуализаторов алгоритмов дискретной математики и создание Интернет-школы программирования автор данной работы совместно с авторским коллективом научно-практической разработки «Инновационная система поиска и подготовки высококвалифицированных специалистов в области производства программного обеспечения на основе проектного и соревновательного подходов» стал лауреатом премии Правительства Российской Федерации в области образования за 2008 год.

СПИСОК ИСТОЧНИКОВ

1. *Федеральный портал «Российское образование»* http://www.edu.ru
2. *Кормен Т., Лайзерсон Ч., Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦМНО, 2000. – 960 с.
3. *Кнут Д.* Искусство программирования. Т. 1: Основные алгоритмы. М.: Вильямс, 2001. – 712 с.
4. *Седжвик Р.* Фундаментальные алгоритмы на C++. Анализ. Структуры данных. Сортировка. Поиск. М.: DiaSoft, 2001. – 687 с.
5. *Ахо А., Хопкрофт Д., Ульман Д.* Структуры данных и алгоритмы. М.: Вильямс, 2010. – 400 с.
6. *Вирт Н.* Алгоритмы и структуры данных. СПб.: Невский Диалект, 2008. – 352 с.
7. *Lawrence A., Badre A., Stassko J.* Empirically evaluating the use of animations to teach algorithms. Technical report. Graphics Visualization and Usability Center, Georgia Institute of Technology, 1993.
8. *Столяр С. Е., Осипова Т. Г., Крылов И. П., Столяр С. С.* Об инструментальных средствах для курса информатики /II Всероссийская конференция «Компьютеры в образовании». СПб., 1994.
9. *Корнеев Г. А., Васильев В. Н., Парфенов В. Г., Столяр С. Е.* Визуализаторы алгоритмов как основной инструмент технологии преподавания дискретной математики и программирования / Труды международной научно-методической конференции «Телематика 2001». СПбГУ ИТМО. 2001, с. 119, 120.
10. *Романовский И. В.* Демонстрационные программы: 1. Решето Эратосфена // Компьютерные инструменты в образовании. 2007. № 1, с. 63–65.

11. *Романовский И. В., Дьяченко В. В.* Демонстрационные программы: 2. Суффиксный массив // Компьютерные инструменты в образовании. 2007. № 2, с. 56–60.
12. *Дискретная математика: алгоритмы.* http://rain.ifmo.ru/cat/
13. *Романовский И. В., Черкасова П. Г.* Пакет демонстрационных программ по курсу дискретного анализа «*DADemo*» // Компьютерные инструменты в образовании. 2002. № 1, с. 55–61.
14. *Удов Г. Г., Шалыто А. А.* Классическая и автоматная реализация визуализатора алгоритма пирамидальной сортировки набора чисел. СПбГУ ИТМО, 2004. http://is.ifmo.ru/vis/pyr/
15. *Прингл Э.* Конечные автоматы в JavaScript. Часть 1. Разработаем виджет. http://www.ibm.com/developerworks/ru/library/wa-finitemach1/
16. *Прингл Э.* Конечные автоматы в JavaScript. Часть 2. Реализация виджета. http://www.ibm.com/developerworks/ru/library/wafinitemach2/wa-finitemac_ru.html
17. *Прингл Э.* Конечные автоматы в JavaScript. Часть 3. Тестируем виджет. http://www.ibm.com/developerworks/ru/library/wa-finitemach3/
18. *Коломейцева О. М., Шалыто А. А.* Преимущества автоматического синтеза программ на языке *JavaScript* по автоматной спецификации (на примере реализации элемента управления *ToolTip*). СПбГУ ИТМО, 2007. http://is.ifmo.ru/projects/java_script/
19. *Андерсон Д.* Дискретная математика и комбинаторика. М.: Вильямс, 2004. – 960 с.
20. *Колмогоров А. Н.* Теория информации и теория алгоритмов. — М.: Наука, 1987. – 304 с.
21. *Sun Oracle Java.* http://www.oracle.com/technetwork/java/index.html
22. *Пратт. Т., Зелковиц М.* Языки программирования: разработка и реализация. СПб.: Питер, 2002. – 688 с.

23. *Агапонов С. В., Джалиашвили З. О., Кречман Д. Л. и др.* Средства дистанционного обучения. Методика, технология, инструментарий. СПб.: БХВ-Петербург, 2003. – 336 с.
24. *Васильев В. Н., Елизаров Р. А., Парфенов В. Г., Столяр С. Е.* Организация дистанционного обучения программистов / Тезисы докладов Всероссийской научно-методической конференции «Телематика'98». СПбГУ ИТМО. 1998, с. 172, 173.
25. *Brown M., Sedgewick R.* A system for Algorithm Animation / Proceedings of the 11th annual conference on Computer graphics and interactive techniques. 1984.
26. *Интернет-школа программирования*. http://ips.ifmo.ru/
27. *Gitta D.* Tutorial on Visualization. http://www.siggraph.org/education/materials/HyperVis/domik/folien.html
28. *Owen G. S.* Visualization Concepts: Overview. http://www.siggraph.org/education/materials/HyperVis/concepts/overview.htm
29. *Complete Collection of Algorithm Animations*. http://www.cs.hope.edu/~alganim/ccaa/
30. *Interactive Data Structure Visualizations*. http://www.student.seas.gwu.edu/~idsv/idsv.html
31. *Jawaa examples*. http://www.cs.duke.edu/csed/jawaa2/examples.html
32. *Sorting Algorithms*. http://www.cs.rit.edu/~atk/Java/Sorting/sorting.html
33. *Rling G., Naps T. L.* A Testbed for Pedagogical Requirements in Algorithm Visualizations / 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE'02). http://ss4sw.org/Publications/2002/TestbedPedReqinAV.pdf
34. *Knowlton K.* L6: Bell Telephone Laboratories Low-Level Linked List Language. 16-minute black-and-white film. N.J.: Murray Hill, 1966.
35. *Knowlton K.* L6: Part II. An Example of L6 Programming. 30-minute black-and-white film, N.J., Murray Hill, 1966.

36. *Stasko J.* Tango: A Framework and System for Algorithm Animation. Technical Report. Brown University Providence, 1989.
37. *Rößling G., Freisleben B.* ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation //Journal of Visual Languages and Computing. Vol. 13. 2002. Issue 3.
38. *Crescenzi P., Demetrescu C., Finocchi I., Petreschi R.* Reversible execution and visualization of programs with Leonardo // Journal of Visual Languages and Computing (JVLC). Vol. 11. 2000. Issue 2.
39. *Brown M.* Zeus: a System for Algorithm Animation / Proceedings of 1991 IEEE Workshop on Visual Languages. 1991.
40. *Italiano G., Cattaneo G., Ferraro U., Scarano V.* CATAI: Concurrent Algorithms and Data Types Animation over the Internet / Proceedings of 15th IFIP World Computer Congress. Vienna, 1998.
41. *Baker J. E., Cruz I. F., Liotta G., Tamassia R.* The Mocha Algorithm Animation System / Proceedings of International Workshop on Advanced Visual Interfaces, 1996.
42. *Корнеев Г. А., Шалыто А. А.* Требования к визуализаторам алгоритмов, выполняемых на базе технологии *Vizi*. http://is.ifmo.ru/vis/req/
43. *Корнеев Г. А.* Автоматизация построения визуализаторов алгоритмов дискретной математики на основе автоматного подхода. Диссертация на соискание ученой степени канд. техн. наук. СПбГУ ИТМО, 2006. http://is.ifmo.ru/disser/korn_disser.pdf
44. *Шалыто А. А.* Технология автоматного программирования // Мир ПК. 2003. № 10, с.74–78. http://is.ifmo.ru/works/tech_aut_prog/
45. *Adobe Flash*. http://www.adobe.com/flashplatform/
46. *Microsoft Silverlight*. http://www.silverlight.net/
47. *Флэнаган Д. JavaScript.* Подробное руководство. СПб.: Символ-Плюс, 2008. – 992 с.
48. *Adobe AIR*. http://www.adobe.com/products/air/
49. *Sun Oracle Java Applets*. http://java.sun.com/applets/

50. *Туккель Н. И., Шалыто А. А., Шамгунов Н. Н.* Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. 2002. № 5, с. 72–99. http://is.ifmo.ru/works/recurse/
51. *Ахо А., Лам М., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии и инструментарий. М.: Вильямс, 2008. – 652 с.
52. *Гамма Э., Хэлм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. – 325 с.
53. *Фаулер М.* Архитектура корпоративных программных приложений. М.: Вильямс, 2004. – 544 с.
54. *Портянкин И.* Swing. Эффектные пользовательские интерфейсы. СПб.: Питер, 2005. – 528 с.
55. *Крейн Д., Бибо Б., Сонневельд Дж.* Ajax на практике. – М.: Вильямс, 2007. – 464 с.
56. *Conway J. H., Guy R. K.* The Book of Numbers. NY: Springer-Verlag. 1996, pp. 127–130.
57. *Решето Эратосфена*. http://ru.wikipedia.org/wiki/Решето_Эратосфена
58. *Шалыто А. А., Туккель Н. И.* Преобразование итеративных алгоритмов в автоматные // Программирование. 2002. № 5, с. 12–26. http://is.ifmo.ru/works/iter/
59. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. – 628 с. http://is.ifmo.ru/books/switch/1
60. *Шалыто А. А., Туккель Н. И.* От тьюрингова программирования к автоматному // Мир ПК. 2002. № 2, с. 144–149. http://is.ifmo.ru/works/turing/
61. *Буч Г., Рамбо Д., Джекобсон* А. Язык UML. Руководство пользователя. М.: ДМК Пресс, 2001. – 432 с.

62. *Шень А.* Программирование: теоремы и задачи. М.: МЦНМО, 2004. – 296 с.
63. *Романовский И. В.* Дискретный анализ. СПб.: Невский диалект, 1999. – 240 с.
64. *Сайт по автоматному программированию*. http://is.ifmo.ru
65. *Добровольский В. А., Степук А. В.* Сортировка подсчетом. СПбГУ ИТМО, 2004. http://is.ifmo.ru/vis/countsort/
66. *Корниенко А. А.* Автоматный подход к реализации элементов графического пользовательского интерфейса. СПбГУ ИТМО, 2004. http://is.ifmo.ru/papers/kornienko/
67. *Wright Th. P.* Learning Curve // Journal of the Aeronautical Sciences, 1936.

Публикации автора

Статьи в журналах из перечня ВАК

68. *Казаков М. А., Шалыто А. А.* Методы построения логики визуализаторов алгоритмов // Открытое образование. 2005, № 4, с. 53–58.
69. *Казаков М. А., Шалыто А. А.* Реализация анимации при построении визуализаторов алгоритмов на основе автоматного подхода // Информационно-управляющие системы. 2005. № 4, с. 51–60. http://is.ifmo.ru/works/visanim/
70. *Казаков М. А., Шалыто А. А.* Использование автоматного программирования для реализации визуализаторов // Компьютерные инструменты в образовании. 2004. № 2, с. 19–33.
71. *Казаков М. А., Шалыто А. А.* Автоматный подход к реализации анимации в визуализаторах алгоритмов // Компьютерные инструменты в образовании. 2005, № 3, с. 52–76.
72. *Казаков М. А., Васильев В. Н., Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Три кита подготовки программистов // Открытые системы. 2009. № 3, с. 54–56.

Другие статьи

73. *Казаков М. А.* Создание системы проведения Интернет-соревнований и дистанционного обучения программированию // Телекоммуникации и информатизация образования. 2002. № 6, с. 81–100.
74. *Казаков М. А., Корнеев Г. А., Шалыто А. А.* Разработка логики визуализаторов алгоритмов на основе конечных автоматов // Телекоммуникации и информатизация образования. 2003. № 6, с. 27–58.

Материалы конференций

75. *Казаков М. А., Столяр С. Е.* Подготовка тестов как часть технологии автоматизированного тестирования / Тезисы докладов XXX науч.-техн. конф. проф.–преподавательского состава. СПбГИТМО (ТУ). 1999, с.105.
76. *Казаков М. А., Осипова Т. Г., Парфенов В. Г., Столяр С. Е.* Интернет-школа программирования в СПбГИТМО (ТУ) / Тезисы докладов Всероссийской научно-методической конференции «Телематика'99». СПбГИТМО (ТУ). 1999, с. 165, 166.
77. *Казаков М. А., Васильев В. Н., Парфенов М. А., Столяр С. Е.* Проблемы подготовки профессиональных программистов и дистанционное обучение в Интернет-школе СПбГИТМО (ТУ) / Материалы II межрегиональной научно-практической конференции «Дистанционное обучение. Проблемы и перспективы взаимодействия вузов Санкт-Петербурга с регионами России». 1999, с. 45–48.
78. *Казаков М. А., Столяр С. Е.* Визуализаторы алгоритмов как элемент технологии преподавания дискретной математики и программирования / Тезисы докладов международной научно-методической конференции «Телематика'2000». СПбГУ ИТМО. 2000, с. 189–191.
79. *Казаков М. А., Шалыто А. А., Туккель Н. И.* Использование автоматного подхода для реализации вычислительных алгоритмов

/ Тезисы докладов международной научно-методической конференции «Телематика'2001». СПбГУ ИТМО. 2001, с.174–176.

80. *Казаков М. А., Мельничук О. П., Парфенов В. Г.* Интернет-школа программирования в СПбГИТМО (ТУ). Реализация и внедрение / Тезисы докладов Всероссийской научно-методической конференции «Телематика'2002». СПбГУ ИТМО. 2002, с. 308, 309.
81. *Казаков М. А., Корнеев Г. А., Шалыто А. А.* Построение логики работы визуализаторов алгоритмов на основе автоматного подхода / Тезисы докладов Всероссийской научно-методической конференции «Телематика'2003». СПбГУ ИТМО. 2003, с. 378, 379.
82. *Казаков М. А.* Использование автоматного подхода для построения визуализаторов / Вестник конференции молодых ученых СПбГУ ИТМО. 2004, с. 166–180.
83. *Казаков М. А.* Система автоматического тестирования программных решений и проведения соревнований в режиме on-line / Вестник конференции молодых ученых СПбГУ ИТМО. 2004, с. 181–189.
84. *Казаков М. А., Шалыто А. А.* Реализация визуализаторов на основе автоматного программирования / Тезисы докладов Всероссийской научно-методической конференции «Телематика'2004». СПбГУ ИТМО. 2004, с. 191, 192.
85. *Казаков М. А., Васильев В. Н., Елизаров Р. А., Парфенов В. Г., Станкевич А. С.* Использование олимпиад и творческих конкурсов при подготовке высококвалифицированных кадров в области информационных технологий / Тезисы докладов Всероссийской научно-методической конференции «Телематика'2004». СПбГУ ИТМО. 2004, с. 437–439.
86. *Казаков М. А., Шалыто А. А.* Технология построения визуализаторов алгоритмов на основе автоматного подхода / Тезисы докладов Всероссийской научно-методической конференции «Телематика'2005». СПбГУ ИТМО. 2005, с. 507–509.

87. *Казаков М. А.* Реализация концепции многоуровневой системы дистанционного обучения на базе Интернет-школы программирования. / Вестник конференции молодых ученых СПбГУ ИТМО. 2005, с. 176–183.
88. *Казаков М. А., Васильев В. Н., Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Инновационная система поиска и подготовки высококвалифицированных разработчиков программного обеспечения на основе проектного и соревновательного подходов / Труды Первого Санкт-Петербургского конгресса «Профессиональное образование, наука, инновации в XXI веке». СПбГУ ИТМО. 2007, с. 84–97.
89. *Казаков М. А., Васильев В. Н., Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Применение проектного подхода на основе автоматного программирования при подготовке разработчиков программного обеспечения / Труды Первого Санкт-Петербургского конгресса «Профессиональное образование, наука, инновации в XXI веке». СПбГУ ИТМО. 2007, с. 98–100.
90. *Казаков М. А., Васильев В. Н., Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Автоматное программирование и проектный подход при подготовке разработчиков программного обеспечения / Труды научно-технической конференции «Научное программное обеспечение в образовании и научных исследованиях». СПбГПУ. 2008, с. 248–250.

Научно-исследовательские работы

91. Министерство образования РФ. НИР № 10038 «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода». Этап 1. «Алгоритмизация программирования задач логического управления». СПбГУ ИТМО. 2000.
92. Министерство образования РФ. НИР № 10038 «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода». Этап 2. «Разработка основных положений

создания программного обеспечения «реактивных» систем». СПбГУ ИТМО, 2001.

93. Министерство образования РФ. НИР № 10038 «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода» Этап 3. «Применение автоматного подхода для программной реализации вычислительных алгоритмов». СПбГУ ИТМО, 2002.
94. Грант РФФИ по проекту 02-07-90114 «Разработка технологии автоматного программирования». Этап 1. «Разработка технологии автоматного программирования для событийных систем». СПбГУ ИТМО, 2002.
95. Министерство образования РФ. НИР № 10038 «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода». Этап 4. «Применение автоматного подхода в объектно-ориентированном программировании». СПбГУ ИТМО, 2003.
96. Грант РФФИ по проекту 02-07-90114 «Разработка технологии автоматного программирования». Этап 2. «Расширение области использования автоматного программирования». СПбГУ ИТМО, 2003.
97. Министерство образования РФ. НИР № 10038 «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода». Этап 5. «Применение методологии создания программного обеспечения систем управления на основе автоматного подхода». СПбГУ ИТМО, 2004.

Приложение. Курсовая работа студентов 3 курса кафедры «Компьютерные технологи» СПбГУ ИТМО А.В. Степука, В.А. Добровольского на тему «Сортировка подсчетом» (2004)

Введение

В работах, приведенных на сайте [1], показана целесообразность применения автоматного программирования для построения визуализаторов.

Визуализатор отображает пошаговое выполнение алгоритма, передвигаясь от состояния к состоянию. Такая логика соответствует работе автомата.

Целью данной работы является построение визуализатора алгоритма сортировки подсчетом на основе автоматного программирования [2].

1. Постановка задачи

Отсортировать за линейное время последовательность, в которой каждый элемент – целое положительное число в известном диапазоне, которое не превосходит заранее известное k.

В реализации визуализатора k равно десяти.

2. Решение задачи

Приведем словесную формулировку решения этой задачи [3].

Идея алгоритма состоит в том, чтобы для каждого элемента x предварительно подсчитать, сколько элементов входной последовательности меньше x. После этого x записывается напрямую в выходной массив в соответствии с этим числом. Например, если семнадцать элементов входного массива меньше x, то в выходном массиве x должен быть записан на место с номером восемнадцать.

Далее в тексте и в самом визуализаторе используются следующие обозначения:

$A[1..n]$ – входная последовательность;

$C[1..k]$ – вспомогательный массив из k элементов;

$B[1..n]$ – выходная отсортированная последовательность.

Работа алгоритма заключается в следующем.

В элемент $C[i]$ заносится число элементов массива A, равных i. Затем находятся частичные суммы последовательности $C[1]$, $C[2]$,..., $C[k]$, каждая из которых является числом элементов, не превосходящих i.

После этого каждый элемент $A[i]$ из входного массива помещается в выходной массив B на позицию, равную значению элемента $C[A[i]]$.

3. Выбор визуализируемых переменных

Перечислим переменные, которые содержат значения, необходимые для визуализации:

$A[]$ – значения входного массива;

$B[]$ – значения выходного массива;

$C[]$ – значения промежуточного массива;

j – текущая позиция в массиве A;

i – текущая позиция в массиве C;

m – текущая позиция в массиве B.

4. Анализ алгоритма для его визуализации

Из рассмотрения алгоритма следует, что его визуализация должна выполняться следующим образом:

- показать массив A, заполненный начальными значениями;
- отразить ход решения задачи – процесс заполнения массива C, основанный на значениях элементов массива A;
- показать подсчет частичных сумм в массиве C;
- отразить процесс последнего этапа решения задачи – заполнения массива B.

5. Реализация алгоритма решения задачи

Приведем алгоритм решения задачи на псевдокоде [3]:

```
1       for i ← 1 to k
2         do C [i] ← 0
3       for j ← 1 to length[A]
4         do C[A[i]] ← C[A[i]] + 1
5       for i ← 2 to k
6         do C[i] ← C[i] + C[i-1]
7       for m ← length[A] downto 1
8         do  B[C[A[m]]] ← A[m]
9           C[A[m]] ← C[A[m]]-1
```

6. Реализация алгоритма на языке *Java*

Перепишем программу на псевдокоде с помощью языка *Java*.

```
1   /**
2    * @param a входной массив
3    * @param K – максимальное значение для элементов A
4    * @return отсортированный массив,
5    */
6   private static Integer[] solve(int K, int[] a) {
7       // Переменные циклов
8       int s, i, j, m;
9       // Количество элементов
10      int N = M.length;
11      // Вспомогательный массив c[]
12      int[] c = new int[K];
13      // Выходной массив b[]
14      int[] b = new int[N];
15      // Заполнение нулями массива c[]
16      for(int s = 0; s < K; s++)
17          c[s] = 0;
18      // Заполнение массива c[]
19      for (j = 0; j < N; j++)
20          c[a[j] - 1] = c[a[j] - 1] + 1;
21      // Подсчет частичных сумм c[]
22      for(i = 1; i < K; i++)
23          c[i] = c[i] + c[i-1];
24      // Формирование выходного массива b[]
25      for(m = N - 1; m >= 0; m--)
26      {
27          b[c[a[m]-1] = a[m];
28          c[a[m] - 1] = c[a[m] - 1] - 1;
29      }
30  }
```

В этой программе операции ввода/вывода не приведены, так как их будет выполнять визуализатор.

7. Построение схемы алгоритма по программе

Построим по тексту программы схему алгоритма (Рис. 1).

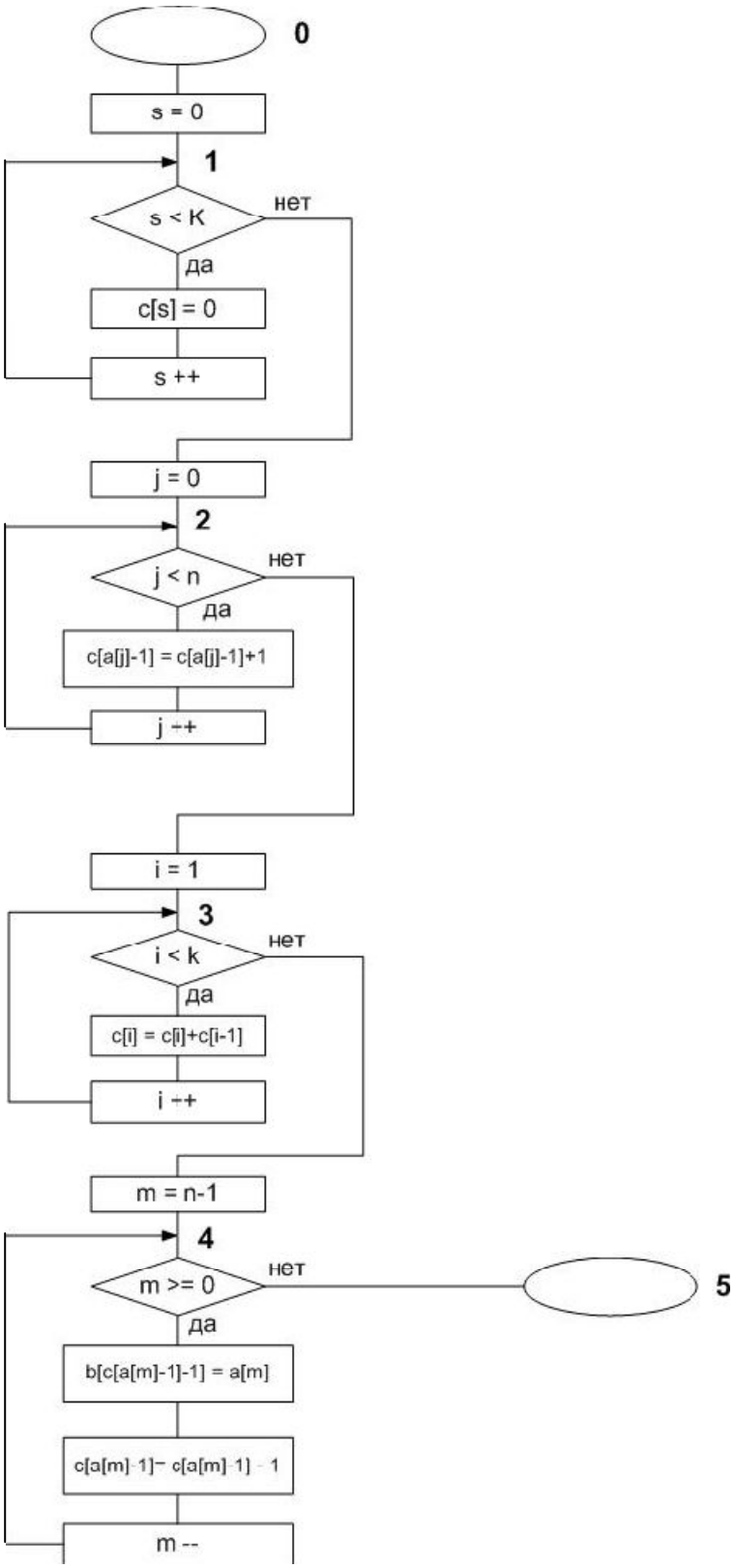

Рис. 1. Схема алгоритма

8. Преобразование схемы алгоритма в граф переходов автомата Мили

В работе [4] показано, что при программной реализации по схеме алгоритма может быть построен, как автомат Мура, так и автомат Мили. В автоматах первого типа действия выполняются в вершинах, а в автоматах второго типа – на переходах. Визуализаторы могут быть построены, как с использованием автоматов Мура, так и автоматов Мили. Для уменьшения числа состояний в настоящей работе применяются автоматы Мили.

В соответствии с методом, изложенным в работе [4], определим состояния, которые будет иметь автомат Мили, соответствующий этой схеме алгоритма. Для этого присвоим номера точкам, следующим за последовательно расположенными операторными вершинами (последовательность может состоять и из одной вершины). Номера присваиваются также начальной и конечным вершинам.

Исходя из изложенного, для схемы алгоритма на Рис. 1 выделим шесть состояний автомата с номерами 0 – 5. Таким образом, автомат визуализации будет иметь шесть состояний, а его граф переходов, который строится за счет определения путей между выделенными точками на схеме алгоритма – шесть вершин (Рис. 2).

9. Формирование набора невизуализируемых переходов

Из анализа графа переходов (Рис. 2) следует, что все переходы будут визуализируемыми.

10. Выбор интерфейса визуализатора

В верхней части визуализатора представляется следующая информация:

- значения массивов *A*[], *B*[], *C*[];
- текущие индексы активных элементов массива.

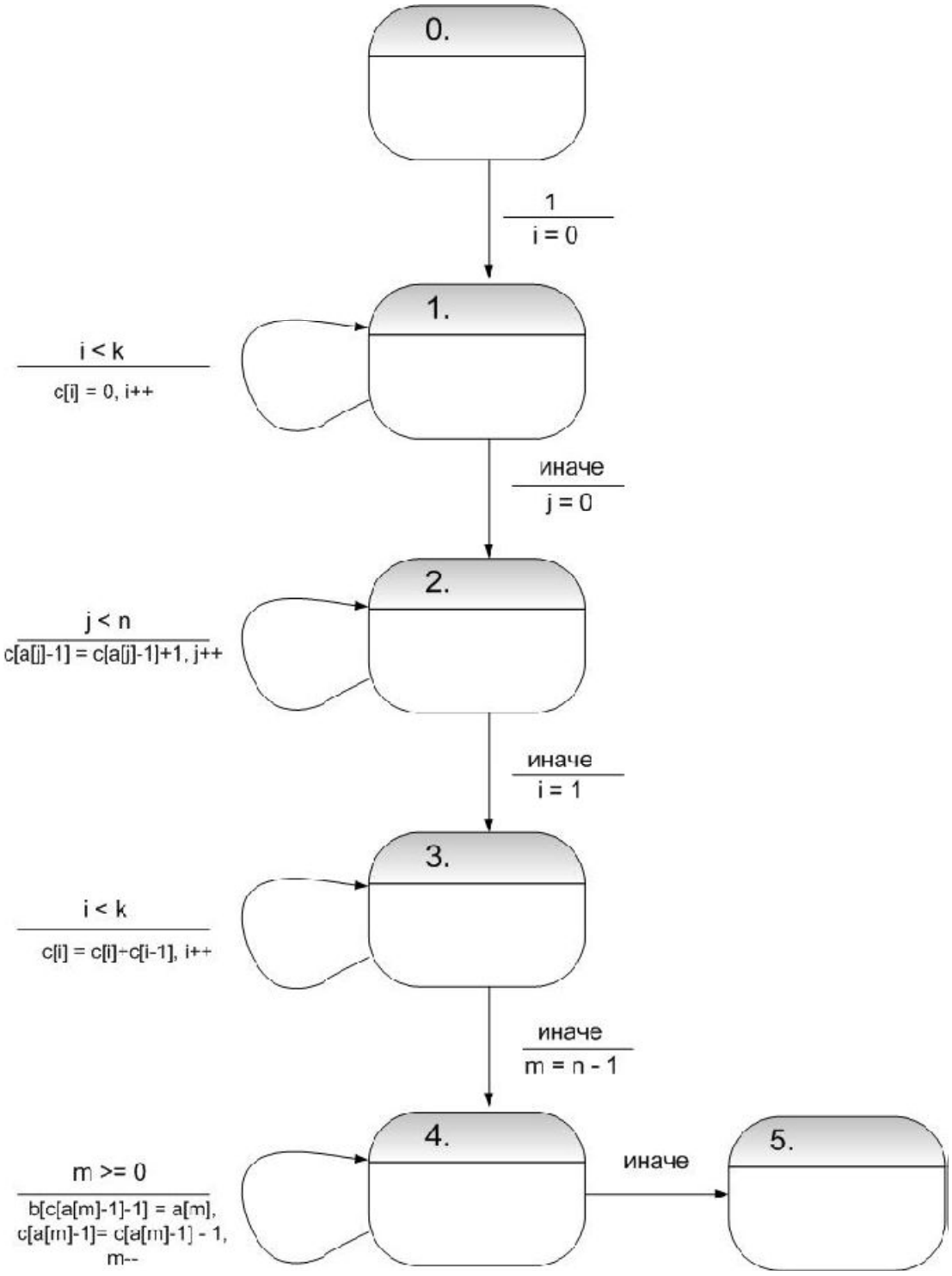

Рис. 2. Граф переходов автомата, реализующего сортировку подсчетом

В нижней части визуализатора расположена панель управления, которая содержит следующие кнопки:

- «>>» – сделать шаг алгоритма;
- «Рестарт» – начать алгоритм заново;

- «Случайно» – начать алгоритм заново с новым произвольным набором данных;
- «Авто» – перейти в автоматический режим;
- «Стоп» – оставить работы в автоматическом режиме и перейти в пошаговый режим;
- «<», «>» – изменение паузы между автоматическими шагами алгоритма.

Помимо кнопок в нижней части расположена надпись «Пауза», отображающая величину задержки в автоматическом режиме.

Среднюю часть визуализатора занимает область комментариев, в которой на каждом шаге отображается описание выполняемого алгоритмом действия.

Визуализатор в исходном состоянии, которое соответствует начальному состоянию автомата, представлен на Рис. 3.

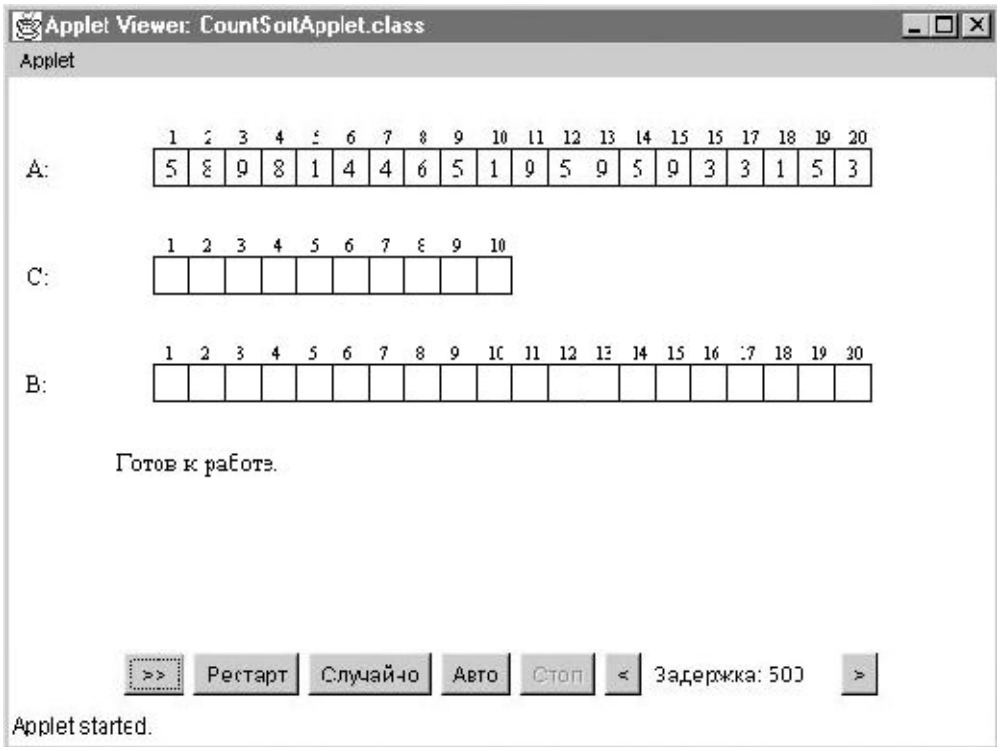

Рис. 3. Состояние 0

11. Сопоставление иллюстраций и комментариев с состояниями автомата

Так как при построении визуализатора используется автомат Мили, то будем в рассматриваемом состоянии отображать действия, которые выполняются при переходе из него. При этом цветами выделяются активные ячейки.

Перечислим состояния автомата визуализации.

Состояние 0. «Готов к работе» (рис.3).

Состояние 1. «Заполнить нулями массив C» (рис.4).

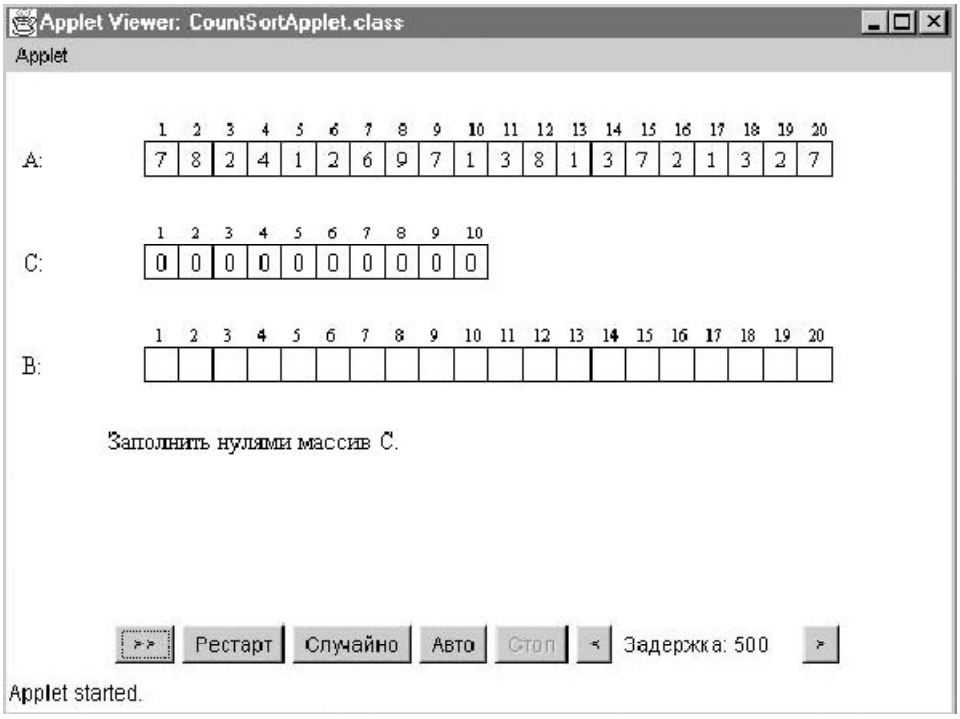

Рис. 4. Состояние 1

Состояние 2. «В массиве C увеличить на единицу $A[j]$-й элемент» (Рис. 5).

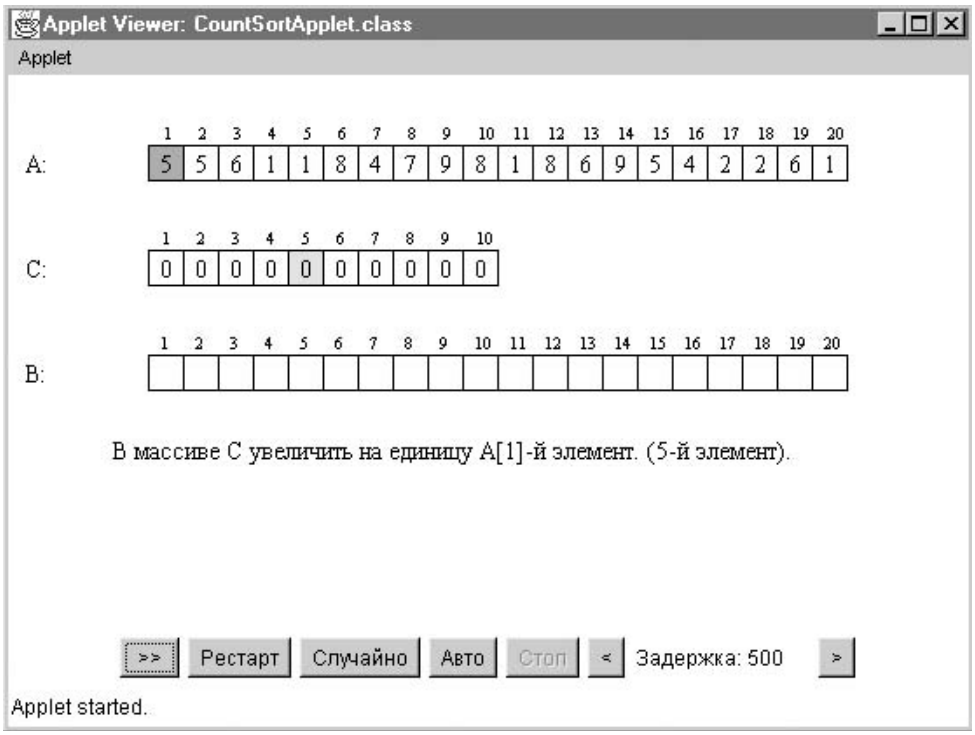

Рис. 5. Состояние 2

Состояние 3. «В *m*-й элемент массива C записывается сумма *m*-го и *m*+1-го элементов» (рис.6).

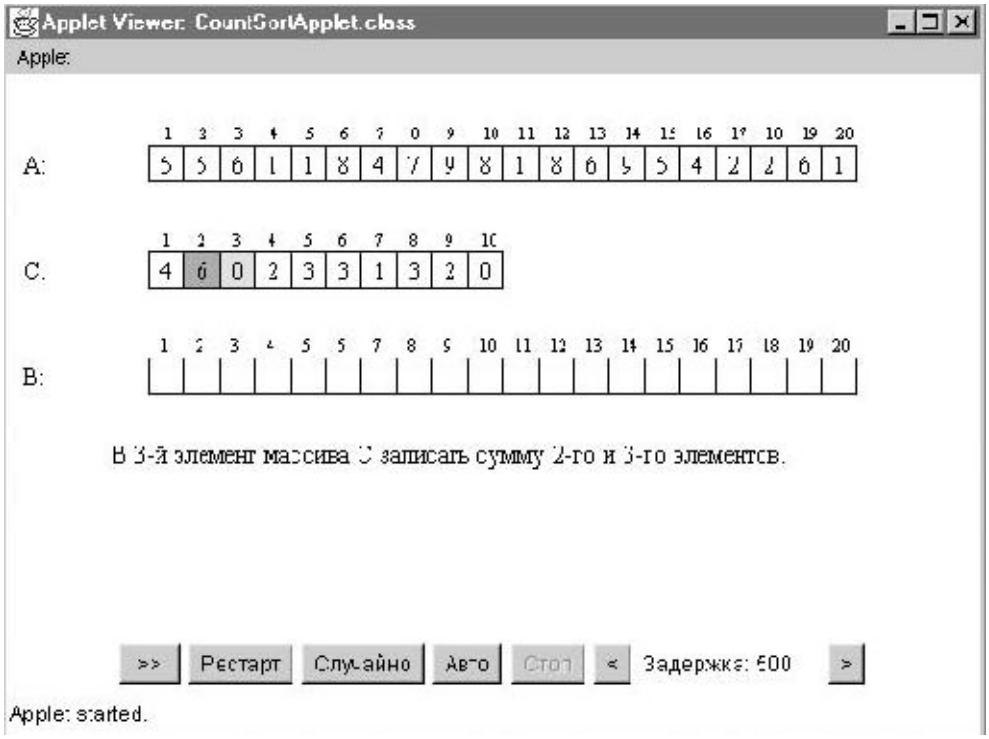

Рис. 6. Состояние 3

Состояние 4. «Найти значение в A[m]-го элемента массива C. Занести A[m] на позицию с этим значением в выходной массив B» (рис.7).

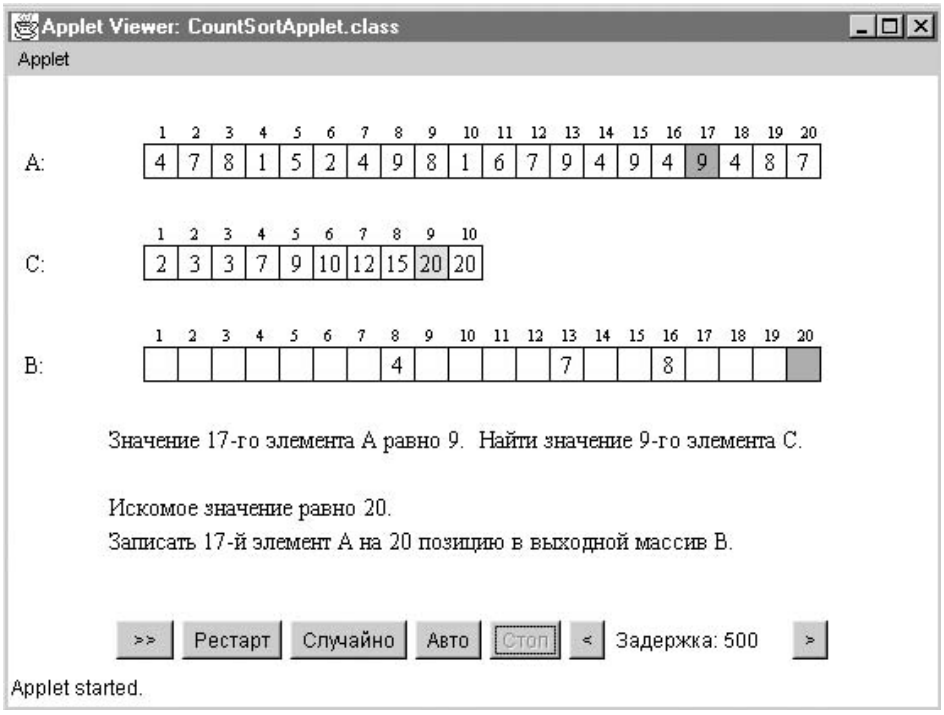

Рис. 7. Состояние 4

Состояние 5. «Массив отсортирован. Работа алгоритма завершена» (рис.8).

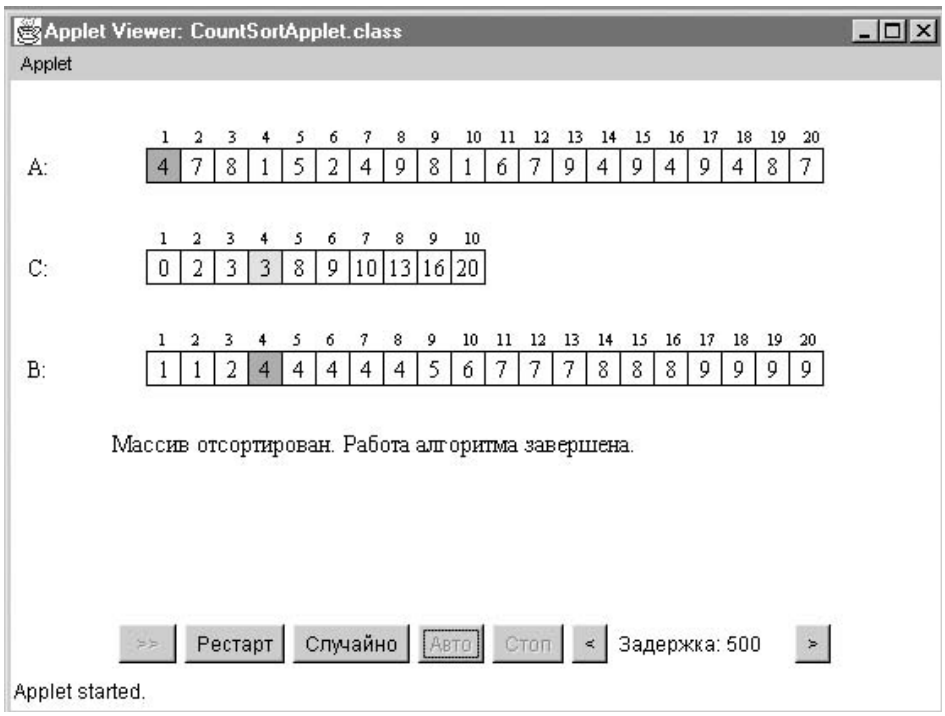

Рис. 8. Состояние 5

12. Выбор архитектуры программы программы визуализатора

Программа состоит из четырех классов:

- *CSControls* – отвечает за действия пользователей;

- *CSCanvas* – реализует работу автомата и отрисовку графики;
- *CountSortApplet* – объединяет части программы в запускаемый апплет;
- *AutoThread* – нить, обеспечивающая выполнение шагов в автоматическом режиме.

Диаграмма классов приведена на Рис. 9.

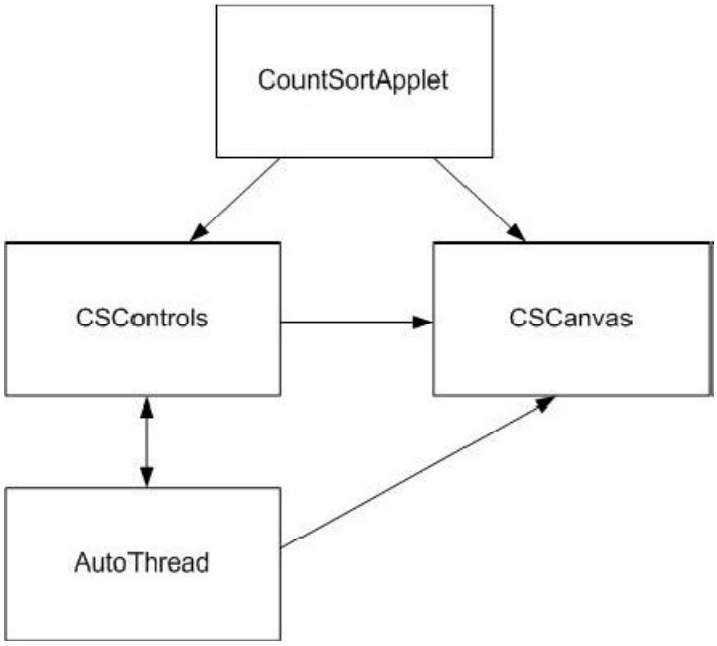

Рис. 9. Диаграмма классов визуализатора

13. Программная реализация визуализатора

В работе [2] было предложено переходить формально и изоморфно от графа переходов автомата к его программной реализации на основе оператора *switch*. Для рассматриваемого примера результатом преобразования графа переходов (Рис. 2) является программа, приведенная ниже:

```
1   switch (state)
2   {
3     case 0:                   // Начальное состояние
4       state = 1; i0 = 0;
5       for (int step = 0; step < N; step++)
6         b[step] = -1;
7       for (int step = 0; step < K; step++)
8         c[step] = 0;
9       break;
10    case 1:                   // Заполнение нулями массива C
11      for (i0 = 0; i0 < K; i0++)
12        c[i0] = 0;
13      state = 2;
14      j = 0;
15      break;
```

```
16    case 2:                  // Поэлементное заполнение массива C
17      if (j < N) {
18        c[a[j]-1] = c[a[j]-1] + 1;
19        j++;
20      }
21      else {
22        state = 3;
23        i = 1;
24      }
25      break;
26    case 3:                  // Преобразование массива C
27      if (i < K) {
28        c[i] = c[i] + c[i-1];
29        i++;
30      } else {
31        state = 4;
32        m = N - 1;
33      }
34      break;
35    case 4:                  // Формирование массива B
36      if (m >= 0){
37        b[c[a[m]-1]-1] = a[m];
38        c[a[m]-1] = c[a[m]-1] - 1;
39        l--;
40      } else {
41        state = 5;
42      }
43      break;
44    case 5:                  // Конечное состояние
45      break;
46  }
```

Такая реализация автомата резко упрощает построение визуализатора, так как к каждому состоянию автомата (метке *case*) могут быть «привязаны» соответствующие иллюстрации и комментарии. В силу того, что реализация визуализатора выполняется с помощью указанного выше паттерна, «привязка» в данном случае осуществляется с помощью второго оператора *switch*. Таким образом, в визуализаторе используется два оператора *switch*, первый из которых реализует автомат, а второй применяется в формирователе иллюстраций и комментариев.

Отметим, что формализованный подход к построению логики визуализатора привел к тому, что отладка логики отсутствовала, а небольших изменений потребовала неформализуемая часть программы, связанная с построением иллюстраций и комментариев.

Полный исходный текст программы визуализатора приведен по адресу http://is.ifmo.ru/vis/countsort/.

Заключение

Использование автоматного подхода для создания визуализаторов было развито в работах на сайте [1].

Данная работа подтверждает успешность этого направления.

Литература

1. Сайт кафедры «Технологии программирования». http://is.ifmo.ru
2. *Казаков М. А., Шалыто А. А.* Использование автоматного программирования для реализации визуализаторов // Компьютерные инструменты в образовании. 2004. № 2.
3. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999.
4. *Шалыто А. А., Туккель Н. И.* Преобразование итеративных алгоритмов в автоматные // Программирование. 2002. № 5, с. 12–26. http://is.ifmo.ru/works/iter/