Н. А. ТЮКАЧЕВ, В. Г. ХЛЕБОСТРОЕВ

C#. АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие

ЛАНЬ
• САНКТ-ПЕТЕРБУРГ • МОСКВА • КРАСНОДАР •
• 2021 •

УДК 004
ББК 32.973я723

Т 98 **Тюкачев Н. А.** C#. Алгоритмы и структуры данных : учебное пособие для СПО / Н. А. Тюкачев, В. Г. Хлебостроев. — Санкт-Петербург : Лань, 2021. — 232 с. : ил. + CD. — Текст : непосредственный.

ISBN 978-5-8114-6817-1

Книга посвящена алгоритмам обработки различных внутренних структур данных — массивов, множеств, деревьев и графов. Кроме того, в отдельной главе дано описание имеющихся в языке C# средств работы с внешними структурами данных — файлами. Описаны основные классы, реализующие методы обработки текстовых и бинарных файлов, организация записи и чтения файлов в режимах последовательного и прямого доступа. На примере алгоритмов сортировки массивов обсуждаются способы оценки эффективности алгоритмов, используемые для их сравнения. Текст содержит большое количество примеров программного кода, способствующих усвоению материала.

Книга предназначена для студентов, обучающихся по направлениям групп специальностей «Информатика и вычислительная техника», «Информационная безопасность», «Электроника, радиотехника и системы связи» среднего профессионального образования, а также учащихся старших классов и лиц, самостоятельно изучающих языки программирования.

УДК 004
ББК 32.973я723

Обложка
П. И. ПОЛЯКОВА

Введение

Учебное пособие посвящено алгоритмам обработки различных внутренних структур данных – массивов, множеств, деревьев и графов. Кроме того, в отдельной главе дано описание имеющихся в языке C# средств работы с внешними структурами данных – файлами. Описаны основные классы, реализующие методы обработки текстовых и бинарных файлов, организация записи и чтения файлов в режимах последовательного и прямого доступа. На примере алгоритмов сортировки массивов обсуждаются способы оценки эффективности алгоритмов, используемые для их сравнения.

Значительное внимание уделено рекурсивным алгоритмам и их сравнению с итерационными аналогами. Возможности рекурсии демонстрируются на примере сложных алгоритмов поиска и оптимизации, известных как *backtracking*-алгоритмы. Далее рекурсивные алгоритмы используются в качестве основного способа обработки деревьев и графов.

Рассматривается один из видов деревьев – двоичные деревья поиска, для которых приводятся алгоритмы добавления, удаления и поиска узлов по заданному ключу. В качестве примеров применения этих алгоритмов рассмотрены сортировка массива с использованием двоичного дерева поиска и синтаксический анализатор. Для визуального представления двоичного дерева поиска и результатов выполнения для него отдельных операций предлагается проект с соответствующим программным кодом.

Глава, посвященная описанию структур графов, содержит большое число алгоритмов их обработки – различных способов обхода, поиска кратчайших путей, построению остова, выделения связных компонент. Проводится сравнительный анализ различных алгоритмов решения для каждой из этих задач. Имеется описание проекта, решающего задачу визуализации процесса построения графа и его обработки.

Последняя глава пособия содержит описание алгоритмической реализации ряда широко используемых численных методов. Текст содержит большое количество примеров программного кода, способствующих усвоению материала. Книга рассчитана на студентов высших учебных заведений, учащихся старших классов, а также лиц, самостоятельно изучающих языки программирования.

Глава 1. Базовые понятия

В этой главе будут подробно рассмотрены некоторые понятия, которые широко используются в практике программирования, однако часто на уровне их интуитивного понимания.

1.1. Данные, типы данных и структуры данных

Информация, зафиксированная и хранящаяся в том или ином виде, называется *данными.* Человек в своей практике использует различные формы представления информации, наиболее употребительными из которых являются текстовая, графическая и звуковая. Соответственно, мы, как правило, имеем дело с текстовыми, графическими или аудиоданными. Но в памяти компьютера любая информация хранится в виде цепочек из нулей и единиц, т.е. в виде двоично-кодированных данных.

Единообразие внутреннего представления информации позволяет упростить физическую реализацию компьютерных операций и, в конечном счете, уменьшить стоимость компьютеров. Однако это приводит к необходимости дополнительно указывать способ интерпретации двоичных данных, хранящихся в памяти компьютера. Этой цели служит концепция *типов данных.* Указывая тип данных, мы тем самым выбираем правило преобразования цепочек из нулей и единиц в привычные нам значения. Например, двоичный код 01011010 может быть интерпретирован как целое число 90 или как буква Z латинского алфавита. Конкретный вид правил интерпретации определяется *реализацией* типов данных, зависящей как от языка программирования, так и от особенностей микропроцессора, обрабатывающего эти данные.

Абстрактным типом данных (АТД) [3] называется некоторое множество *значений* и множество *операций* над этими значениями, заданные без указания способа их реализации. Так, целочисленный тип данных можно определить как АТД, значения которого образуют некоторое подмножество целых чисел, а в качестве операций использоваться сложение, вычитание, умножение и целочисленное деление. При этом не уточняется ни вид используемого подмножества целых чисел, ни способ их записи в виде двоичных кодов, ни детали выполнения операций над ними.

Под данными обычно понимают неделимые единицы информации, такие как целое или вещественное число, отдельный символ. В языках программирования с ними связывают *простые* типы данных [18]. Перечень таких типов для языка C# приведен в главе 2 книги «C#. Введение в программирование». Однако для решения практических задач необходимо использовать информационные модели объектов реального мира. Для построения таких моделей недостаточно простых

данных: требуется создавать на их основе более сложные *структуры данных*. Базовыми структурами данных являются [30, 31]:

- массивы,
- списки,
- стеки,
- очереди,
- деревья,
- графы.

Каждая структура данных характеризуется способом доступа к составляющим ее элементам. Именно доступ к элементам, а также изменение структуры путем добавления и удаления отдельных элементов являются основными операциями над любой структурой данных. Поэтому для структур данных можно ввести понятие *составного абстрактного типа данных*. Определение составного АТД должно включать указание на тип(ы) элементов соответствующих структур данных и перечисление операций над ними.

Для работы со структурами данных необходимо иметь *реализацию* составных АТД. В языке C# такие реализации представляются в виде классов, содержащих элементы структур данных в виде полей, а реализации операций – в виде методов.

1.2. Алгоритмы, анализ алгоритмов

Как известно, алгоритм представляет собой описание способа решения некоторой задачи [12-14]. Но обычно для решения одной и той же задачи может быть предложено несколько способов, т.е. алгоритмов. Соответственно возникает проблема выбора наилучшего из них. Такой выбор возможен, если определены оцениваемые характеристики и степень значимости каждой из них.

Главной оцениваемой характеристикой алгоритма является, очевидно, обеспечиваемая им скорость решения задачи (временн***а***я эффективность) [27]. Вторая важная характеристика – это сложность реализации. Очевидно, не при всех обстоятельствах следует отдавать предпочтение более быстрому, но и более сложному алгоритму. Если скорость выполнения не является критически важной, то менее сложный в реализации алгоритм упростит процесс отладки из-за меньшей вероятности возникновения ошибки. Наконец, некоторые алгоритмы для своей реализации могут требовать использования дополнительного объема памяти для хранения промежуточных данных.

Каким же образом сравнить алгоритмы по скорости выполнения. Наиболее очевидный способ – реализовать их и выполнить на одном и том же компьютере с одним и тем же набором данных. Но сразу же возникает вопрос: зачем мне реализовывать несколько алгоритмов, если в конечном счете нужен только один? Кроме того, где гарантия, что выбранный для тестирования набор данных не является в каком-то смысле «плохим» для некоторых

алгоритмов. Использование же нескольких тестовых наборов делает процедуру выбора неприемлемо трудной.

Поэтому для анализа алгоритмов используют теоретические методы, главным из которых является метод *асимптотической оценки временной эффективности* [2, 27]. Суть этого метода заключается в установлении функциональной зависимости числа выполняемых алгоритмом действий от объема исходных данных в предельном случае больших объемов этих данных. Уточним понятия выполняемого алгоритмом действия и объема исходных данных, называемого также *размерностью задачи*.

Будем исходить из того, что алгоритм записан на языке C#. Тогда под действием алгоритма будем понимать выражение, не содержащее вызова методов. Такие выражения содержат небольшое число операций и время их вычисления (выполнения действия алгоритма) можно считать примерно одинаковым. Что касается объема исходных данных, то поскольку речь в основном будет идти о работе со структурами данных, в качестве такового естественно выбрать число элементов в структуре.

Рассмотрим в качестве примера алгоритм поиска наибольшего значения в массиве, содержащем N целых чисел. В этом случае размерностью задачи является размер массива N.

Листинг 1.1. Поиск максимального элемента в массиве

```
int max = a[0];      // повторяется 1 раз
int i_max = 0;       // повторяется 1 раз
for(int i = 1; i < N; i++) // повторяется N раз
    if (a[i] > max) // повторяется N - 1 раз
    {
        max = a[i];        // повторяется m раз
        i_max = i;         // повторяется m раз
    }
```

Число повторений двух последних действий m зависит от значений элементов массива. Очевидно, что $0 \le m < N$. Суммируя число повторений каждого действия алгоритма и обозначая среднее время выполнения одного действия t, получим следующее значение времени работы алгоритма:

$$T(N) = 2t + Nt + (N-1)t + 2mt = (2N + 2m - 1)t. \qquad (1.1)$$

При анализе алгоритмов оценку их эффективности делают, исходя из предположения о наихудшем с точки зрения времени выполнения сочетании

значений исходных данных. Обоснованность такого подхода очевидна: худшего результата алгоритм никогда не покажет, но может показать гораздо лучший.

Для рассмотренного алгоритма наихудшим является случай, когда массив упорядочен по возрастанию и, соответственно, $m = N - 1$. Тогда $T(N) = (4N - 3)t$, т. е. время работы алгоритма является линейной функцией от размерности задачи.

Временная эффективность алгоритмов может выражаться полиномами второй и более высоких степеней от размерности задачи. Во всех случаях такой полиномиальной зависимости переход к пределу больших n заключается в отбрасывании всех членов полинома, кроме старшего. В результате мы приходим к понятию *скорости роста (rate of growth)* для алгоритма, т. е. закону изменения времени выполнения алгоритма от размерности задачи. Для обозначения скорости роста алгоритмов используется так называемая *O*-нотация, которая определяется следующим образом [27, 40].

Определение.

$O(g(N))$ – это множество функций $f(N)$ таких, что существуют константы c и N_0, что $0 \le f(N) \le c \cdot g(N)$ для всех $N > N_0$.

Таким образом, *O*-нотация используется для обозначения *асимптотического верхнего предела* для некоторого множества функций. Применительно к скорости роста алгоритма в качестве таких функций выступают зависимости среднего времени выполнения алгоритма от размерности задачи. Рисунок 1.1 иллюстрирует понятие *O*-нотации. На нем по оси ординат отложено время выполнения алгоритма, а по оси абсцисс – размерность задачи. Штриховой вертикальной линией показано граничное значение N_0, с которого начинает выполняться неравенство $f(N) \le c \cdot g(N)$.

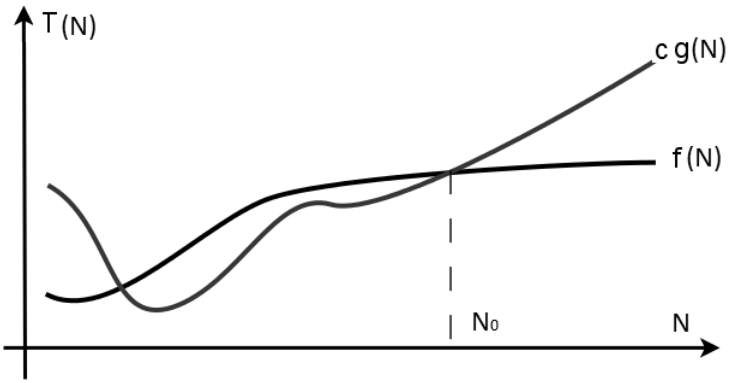


Рис. 1.1. Иллюстрация понятия *O*-нотации

Использование *O*-нотации избавляет от необходимости при анализе алгоритма включать в рассмотрение детали его реализации (язык программирования, среду выполнения, технические особенности компьютера). Выражение «скорость роста алгоритма равна *O(g(N))*» не зависит от входных данных и полезно для распределения алгоритмов по категориям, независимо от входных данных и деталей реализации.

Зная скорость роста алгоритма, можно прогнозировать, во сколько раз увеличится время выполнения алгоритма при кратном увеличении размерности задачи. В таблице 1.1 показано, как сказывается удвоение размерности задачи на времени выполнения алгоритмов с различными скоростями роста [40].

Таблица 1.1.

Изменение времени выполнения алгоритма

Скорость роста	**Степень влияния на время выполнения**
$O(1)$	Нет влияния
$O(\log_2 N)$	Небольшой рост
$O(N)$	Удвоение
$O(N \cdot \log_2 N)$	Немного больше удвоения
$O(N^{3/2})$	Почти в 3 раза
$O(N^2)$	Увеличение в 4 раза
$O(N^3)$	Увеличение в 8 раз
$O(2^N)$	Увеличение в 2^N раз

В теории алгоритмов принято деление задач на [27]:

- задачи класса P, для которых существуют алгоритмы со скоростью роста $O(N^k)$, т.е. с полиномиальным временем выполнения;
- задачи класса NPC, для которых не существует алгоритмов с полиномиальным временем выполнения, так называемые NP-полные задачи.

Подавляющее большинство практически значимых задач относится к первому классу, причем с невысокой степенью *k*.

1.3. Измерение времени выполнения программного кода

Зависимость времени выполнения алгоритма от размерности задачи может быть проверена экспериментально. Платформа `.Net Framework`

предоставляет для этого достаточно развитый инструментарий. Это, прежде всего, структура *TimeSpan*, содержащая набор свойств для хранения временных промежутков в днях, часах, минутах, секундах и тактах (тиках). Именно этот тип данных будет использоваться далее для всех переменных, хранящих временные интервалы.

Мы рассмотрим два возможных подхода к решению проблемы оценки времени выполнения программы или отдельной части программного кода: с помощью объекта класса *Stopwatch* и с помощью непосредственного измерения на уровне потока выполнения.

1.3.1. Измерение с помощью объекта класса Stopwatch

Класс *Stopwatch* объявлен в пространстве имен *System.Diagnostics* и содержит средства работы с временными интервалами. Объект этого класса может быть использован в качестве таймера, который запускается и останавливается для фиксации временн***о***го промежутка. Класс имеет единственный конструктор без параметров. Основными методами класса являются методы:

- *Start*() – запускает процесс измерения времени;
- *Stop*() – останавливает процесс измерения времени;
- *Restart*() – перезапускает процесс измерения времени.

Для хранения значения временного промежутка, прошедшего между моментами запуска и остановки процесса измерения времени используется свойство *Elapsed* типа *TimeSpan* или его вариант *ElapsedMilliseconds*. Схема измерения времени выполнения программного кода с помощью объекта класса *Stopwatch* представлена в листинге 1.2.

Листинг 1.2. Измерение времени выполнения с помощью Stopwatch

```
const int n = 10000;
static void Main(string[] args)
{
   int[] a = new int[n];
   Random rnd = new Random();
   for (int i = 0; i < n; i++)
      a[i] = rnd.Next() % 500;
   Stopwatch stpWatch = new Stopwatch();
   stpWatch.Start();
```

```
        // здесь тестируемый код
        stpWatch.Stop();
        Console.WriteLine("StopWatch: " +
    stpWatch.ElapsedMilliseconds.ToString());
        Console.ReadLine();
    }
```

Описанный способ измерения времени выполнения программного кода имеет один существенный недостаток: предполагается, что в течение всего времени измерения процессор выполняет только этот код. Однако в многозадачных средах современных операционных систем это далеко не так. Далее будет описан другой способ измерения времени выполнения, свободный от указанного недостатка [40].

1.3.2. Измерение на уровне потока выполнения. Класс Timing

Для загруженной на выполнение программы операционная система выделяет необходимые ресурсы (защищенную от доступа других процессов область оперативной памяти), создавая так называемый *процесс*. К числу этих ресурсов относится защищенная от других процессов область оперативной памяти, средства работы с файлами, часть процессорного времени. Для доступа к центральному процессору внутри процесса создается один или несколько *потоков управления*. В пределах каждого потока управления выполняется некоторый программный код. Центральный процессор по определенному алгоритму переключается между разными потоками выполнения, имитируя одновременный характер выполнения разных частей программы. Кроме того, часть потоков могут выполнять те или иные служебные действия, предписанные операционной системой.

Одним из наиболее известных действий такого рода является *сбор мусора (garbage collection)*. Так называется процесс удаления из динамической памяти ставших ненужными объектов. Этот процесс время от времени инициируется операционной системой и может прервать выполнение интересующей нас программы. Тогда значение измеренного времени выполнения окажется завышенным. Решение проблемы заключается в том, чтобы выполнить сбор мусора непосредственно перед началом измерения. Среда *.Net Framework* предоставляет специальный объект *GC*, с помощью которого можно обратиться к методам управления процессом сбора мусора. Так, инициировать процесс можно, обратившись к методу *Collect*(). Надо еще учесть, что удаление объекта может сопровождаться выполнением так называемого *финализатора* – метода, содержащего необходимые завершающие действия, например, закрытие файлов, с которыми работал удаляемый

объект. Метод *WaitForPendingFinalizers*() приостанавливает выполнение всех потоков до завершения работы финализаторов удаляемых объектов.

Вернемся теперь непосредственно к проблеме измерения времени выполнения алгоритма. Реализующий его программный код выполняется одним из потоков управления в рамках процесса. Системный класс *Process* предоставляет средства работы с процессами, в частности, средства доступ к потокам через индексированное свойство *Threads*. Обращение к *i*-му потоку текущего (т.е. выполняющего данную программу) процесса имеет вид: *Process.GetCurrentProcess.Threads*[*i*]. Наконец, текущее время (как объект класса *TimeSpan*) в рамках конкретного потока выполнения можно получить, обратившись к свойству *UserProcessorTime*.

Объединим все сказанное и создадим класс *Timing*, подобный классу *Stopwatch*, но с методами, более точно измеряющими время выполнения программного кода. Этот класс представлен в листинге 1.3.

Листинг 1.3. Класс Timing

```
class Timing
{
   TimeSpan duration;
   TimeSpan[] threads;

   public Timing()
   {
      duration = new TimeSpan(0);
      threads = new Time-
Span[Process.GetCurrentProcess().
      Threads.Count];
   }

   public void StartTime()
   {
      GC.Collect();
      GC.WaitForPendingFinalizers();
```

```
        for (int i = 0; i < threads.Length; i++)
            threads[i] = Proc-
ess.GetCurrentProcess().Threads[i].
                        UserProcessorTime;
    }

    public void StopTime()
    {
        TimeSpan tmp;
        for (int i = 0; i < threads.Length; i++)
        {
            tmp = Proc-
ess.GetCurrentProcess().Threads[i].
             UserProcessorTime.Subtract(threads[i]);
            if (tmp > TimeSpan.Zero)
                  duration = tmp;
         }

    }

    public TimeSpan Result()
    {
        return duration;
    }
}
```

Класс *Timing* имеет два поля: *duration* – для хранения результата измерения и поле-массив *threads*, в котором хранятся значения времени, соответствующие началу измерения для всех потоков процесса. Необходимость наличия такой информации связана с тем, что заранее невозможно сказать, какому из потоков процесса будет поручено выполнение интересующего нас программного кода. Инициализация массива *threads* осуществляет-

ся в методе *StartTime* после вызова сборщика мусора. В методе *StopTime* производится повторный запрос текущего времени для всех процессов и выбирается тот из них, у которого результат оказывается отличным от исходного значения. Именно этот процесс и выполнял исследуемый код.

Листинг 1.4 содержит код функции, в которой используются два способа измерения времени выполнения одного и того же алгоритма сортировки.

Листинг 1.4. Сравнение двух способов измерения времени выполнения алгоритма

```
static void Main(string[] rags)
{
   int[] a = new int[n];
   Random rend = new Random();
   for (int I = 0; I < n; i++)
      a[i] = rnd.Next() % 500;
   Timing objT = new Timing();
   Stopwatch stpWatch = new Stopwatch();
   objT.StartTime();
   stpWatch.Start();
   SortInsertion(a);
   stpWatch.Stop();
   objT.StopTime();
   Console.WriteLine("StopWatch: " +
stpWatch.Elapsed.ToString());
   Console.WriteLine("Timing:    " +
objT.Result().ToString());
   Console.ReadLine();
}
```

На рисунке 1.2 показан результат этого сравнения для массива из 100 тысяч целочисленных значений на процессоре с тактовой частотой в 2.4 ГГц.

```
StopWatch: 00:00:16.7604719
Timing:    00:00:16.1562500
```

Рис. 1.2. Сравнение двух способов измерения времени выполнения алгоритма

Видно, что время, полученное вторым способом, заметно меньше измеренного средствами класса `Stopwatch`, что подтверждает факт влияния системных процессов на результат измерения.

Выводы

Существует взаимосвязь между способом представления данных для решаемой задачи и алгоритмом ее решения. Для представления данных могут быть использованы различные структуры данных (массивы, списки, деревья и т. д.) [47]. Если рассматривать эти структуры на абстрактном уровне, как АДТ, очевидно, что для всех этих структур существует похожий набор операций, например, добавление, удаление и поиск элементов. Однако реализация этих операций может существенно отличаться для разных структур данных. Более того, даже для одной и той же структуры данных, зачастую, могут быть предложены разные алгоритмы выполнения операций. В данной главе это было продемонстрировано на примере операций сортировки и поиска в массивах.

Наличие нескольких алгоритмов решения одной и той же задачи ставит вопрос о выборе лучшего из них. Наиболее часто используемым критерием выбора является скорость роста алгоритма – зависимость среднего времени его выполнения от размерности задачи, в качестве которой обычно выступает число элементов в обрабатываемой структуре данных (например, размер массива). Для обозначения скорости роста алгоритма используется так называемая *O*-нотация. Большинство практически значимых алгоритмов имеют скорость роста не выше $O(N^k)$ с небольшим значением k.

Существует возможность экспериментальной проверки закона скорости роста алгоритма. Для этой цели можно использовать .`Net`-классы `Stopwatch` и `Process`.

Упражнения

1. Для каких значений N справедливо $10 \times N \times log_2 N > N^2$?
2. Для каких значений N выражение $N^{3/2}$ имеет значение в пределах от $N \times (log_2 N)^2/2$ до $2 \times N \times (log_2 N)^2$?
3. Докажите, что $[log_2 N]+1$ – это количество бит, необходимое для представления числа N в двоичной форме ($[log_2 N]$ – целая часть логарифма).
4. Покажите, что $N \times log_2 N = O(N^{3/2})$.

Глава 2. Алгоритмы поиска и сортировки

В разделе описываются основные алгоритмы поиска и сортировки. Задача поиска заключается в установлении факта наличия в той или иной структуре данных принадлежащих ей элементов, которые обладают некоторым наперед заданным свойством. Это свойство называется *ключом поиска.* Как правило, задача поиска входит составной частью в более общую задачу, требующую доступа к искомому элементу.

Сортировкой некоторой структуры данных называют процесс перестановки ее элементов в определенном *порядке*. Для сортировки выбирают один из параметров, характеризующих эти элементы. Такой параметр называют *ключом сортировки*. Например, в качестве ключа сортировки списка сотрудников может использоваться фамилия сотрудника, а может – год его рождения.

Цель сортировки заключается в том, чтобы облегчить последующий поиск элементов в отсортированной структуре данных. Поэтому элементы сортировки и поиска присутствуют почти во всех задачах обработки информации.

В этой главе мы будем рассматривать алгоритмы поиска и сортировки на примере одномерных целочисленных массивов. Этого достаточно, чтобы продемонстрировать главные особенности этих алгоритмов. Ключом как поиска, так и сортировки будем считать значение элемента массива.

2.1. Алгоритмы поиска

Как уже говорилось выше, поиск в отсортированном массиве происходит гораздо быстрее, чем в массиве неотсортированном. В этих двух случаях используются разные алгоритмы поиска. Кроме того, массив изначально может заполняться так, чтобы сделать поиск в нем максимально простым. Ниже будет описан способ такого заполнения, получивший название *хеширование*.

2.1.1. Поиск в неупорядоченном массиве

Простейшим методом поиска элемента, находящегося в неупорядоченном массиве, является последовательный просмотр каждого элемента массива. Перебор элементов заканчивается либо тогда, когда найдется элемент с искомым ключом, либо когда будет достигнут конец массива – это значит, что такого элемента в массиве нет.

Результатом поиска желательно иметь индекс искомого элемента, если элемент найден. Поэтому алгоритм, осуществляющий поиск элемента в массиве, оформим в виде статического метода целого типа. Если элемент найден, возвращаемое значение равно индексу первого найденного элемента. Если же искомого элемента в массиве нет, то метод возвращает значение, равное -1. Алгоритм простого поиска элемента *x* в массиве *a* приведен в листинге 2.1.

Листинг 2.1. Простой поиск

```
static int SearchSimple(int[] a, int x)
{
   int L = a.Length;
   int i = 0;
// с проверкой выхода за границу массива
   while (i < L && a[i] != x)
      i++;
   if (i < L)
// если элемент найден, возвращаем его индекс
      return i;
   else
// если элемент не найден, возвращаем -1
      return -1;
}
```

В этом алгоритме выход из цикла осуществляется по двум условиям: элемент найден или достигнут конец массива. Проверку выхода за границу массива можно опустить, если искомый элемент гарантированно находится в массиве. Такой гарантией может служить *барьер* – дополнительный элемент массива, значение которого равно искомому элементу. Установка барьера производится до цикла поиска.

Листинг 2.2. Поиск с барьером

```
static int SearchBarrier(int[] a, int x)
{
   int L = a.Length;
```

```
// увеличиваем размер массива на 1
   Array.Resize<int>(ref a, ++L);
   a[L - 1] = x;  // устанавливаем барьер
   int i = 0;
// без проверки выхода за границу массива
   while (a[i] != x)
      i++;
   if (i < L - 1)
      return i;
   else
      return -1;
}
```

В этом примере для изменения размера массива использован обобщенный статический метод *Resize* класса *Array* из пространства имен *System*. Благодаря наличию барьера в условии продолжения для цикла отсутствует проверка выхода за границу массива.

В алгоритмах поиска основной выполняемой операцией является сравнение. Очевидно, что при последовательном поиске в среднем требуется $(N+1)/2$ сравнений. Таким образом, данный алгоритм характеризуется линейной функцией скорости роста – *O(N)*.

2.1.2. Поиск в упорядоченном массиве

Поиск можно значительно ускорить, если массив упорядочен, например, по возрастанию. В этом случае чаще всего применяется *метод деления пополам* или *бинарный поиск*. Суть этого метода заключается в следующем. Сначала искомый элемент сравнивается со средним элементом массива. Если искомый элемент больше среднего, то поиск продолжается в правой части массива, если меньше среднего – то в левой части. При каждом сравнении из рассмотрения исключается половина элементов – не имеет смысла искать элемент больше среднего в левой части, содержащей меньшие значения.

Максимальное число требующихся сравнений равно log_2N. Алгоритм приведен в листинге 2.3.

Листинг 2.3. Бинарный поиск

```
static int SearchBinary(int[] a, int x)
```

```
{
   int m, left = 0, right = a.Length - 1;
   do
   {
      m = (left + right) / 2;
      if (x > a[m])
         left = m + 1;
      else
         right = m - 1;
}
   while ((a[m] != x) && (left <= right));
   if (a[m] == x)
      return m;
   else
      return -1;
}
```

Данный алгоритм по сравнению с предыдущим обладает гораздо более высокой скоростью выполнения. Его временная эффективность характеризуется скоростью роста $O(log_2N)$.

2.2. Алгоритмы сортировки

Основным требованием к методам сортировки массивов является экономное использование памяти. Это означает, что переупорядочение элементов необходимо выполнять *in situ* (на том же месте). Поэтому основным критерием эффективности алгоритма сортировки является его быстродействие. При сортировке элементов в массиве выполняются две основных операции: сравнения элементов по ключу сортировки и пересылка элементов. И число сравнений, и число перестановок зависят от размера массива N.

Хорошие алгоритмы сортировки требуют порядка $N*log_2N$ сравнений, более простые – порядка N^2 сравнений ключей. Как правило, усовершенствованные алгоритмы сортировки содержат меньшее число операций, однако это достигается путем усложнения самих операций. Поэтому простые методы являются предпочтительными при малых N, но их не рекомендуется использовать для сортировки массивов больших размеров.

При выборе того или иного алгоритма сортировки для конкретной задачи необходимо учитывать ряд условий. Перечислим наиболее важные из них.

Исходная упорядоченность массива. В исходном массиве могут встречаться упорядоченные участки. В предельном случае массив может оказаться уже упорядоченным. Одни алгоритмы не учитывают исходной упорядоченности и требуют одного и того же времени для сортировки любого множества данного объема, другие выполняются тем быстрее, чем лучше упорядоченность на входе. Говорят, что сортировка демонстрирует *естественное поведение*, если число сравнений и число пересылок имеют наименьшие значения из возможных, в случае упорядоченного массива и возрастают с ростом неупорядоченности, и *неестественное поведение* в противном случае.

Временные характеристики операций. При определении алгоритма время выполнения обычно считается пропорциональным числу сравнений ключей. Ясно, однако, что сравнение числовых ключей выполняется быстрее, чем строковых; операции пересылки выполняются тем быстрее, чем меньше объем записи, и т.п. В зависимости от структуры сортируемых элементов может быть выбран алгоритм, обеспечивающий минимизацию числа тех или иных операций.

Метод сортировки называется *устойчивым*, если относительный порядок элементов с одинаковыми ключами не меняется при сортировке. Устойчивость сортировки часто бывает желательна, если элементы уже упорядочены по одному ключу, а сортировка ведется по другому ключу.

Методы сортировки массива можно разбить на три основных класса в зависимости от лежащего в их основе приема:

- сортировка выбором;
- сортировка включениями;
- сортировка обменом.

В дальнейшем изложении предполагается, что результатом сортировки является массив, элементы которого упорядочены по возрастанию ключа. При этом ключом элемента считается значение самого элемента.

2.2.1. Сортировка простым выбором

Это простой и наиболее очевидный способ сортировки. Соответствующий алгоритм можно описать следующим образом.

1. Среди элементов массива выбирается элемент с наименьшим значением ключа.
2. Выбранный элемент меняется местами с первым элементом массива.

После этого массив можно рассматривать как состоящий из двух частей: левой – уже отсортированной, и правой – неотсортированной. Повторное применение шагов 1 и 2, но уже к неотсортированной части массива при-

ведет к ее уменьшению на один элемент и, соответственно, к увеличению на один элемент отсортированной части массива. Очевидно, что сортировка всего массива требует чтобы действия 1 и 2 были выполнены $N - 1$ раз.

Функция, реализующая алгоритм сортировки простым выбором, представлена в листинге 2.4. Эта функция имеет только один параметр – сортируемый массив.

Листинг 2.4. Сортировка простым выбором

```
public void SortSelection(int[] a)
{
    int N = a.Length;
    int min = 0, imin = 0, i;
    for (i = 0; i < N - 1; i++)
    {
        min = a[i]; imin = i;
        // в этом цикле ищем минимальный элемент
        for (int j = i + 1; j < N; j++)
            if (a[j] < min)
            {
                min = a[j]; imin = j;
            }
        if (i != imin)
        {
// добавление нового элемента в отсортированную
//часть
            a[imin] = a[i];
            a[i] = min;
        }
    }
}
```

Обычно обмен местами двух элементов массива выполняется в три действия с использованием вспомогательной переменной. В данном примере роль вспомогательной переменной играет переменная min. Это несколько уменьшает время работы программы, так как доступ к простой переменной осуществляется быстрее, чем к элементу массива.

Возможна довольно простая модификация данного алгоритма, предусматривающая поиск в одном цикле просмотра одновременно минимума и максимума с последующим их обменом с первым и последним элементами неотсортированной части массива соответственно.

Скорость роста для данного алгоритма может быть определена, если найти зависимость числа *C* сравнений элементов и числа *M* их перестановок от размера массива. Число сравнений не зависит от исходной упорядоченности массива и определяется формулой

$$C = \sum_{i=N-1}^{1} i = \frac{N(N-1)}{2} \sim \frac{N^2}{2} \tag{2.1}$$

в пределе больших *N*.

В случае изначальной упорядоченности массива по возрастанию число перестановок минимально и $M = N - 1$. В случае, если изначально массив упорядочен по убыванию, число перестановок максимально и равно:

$$M = \sum_{i=0}^{(N-n)/2} (N-2i) + 3(N-1) = \frac{N^2 - 14N - n}{4}. \tag{2.2}$$

В этой формуле $n = 3$ для нечетных *N* и $n = 4$ для четных *N*. Детальный анализ, выполненный Д. Кнутом [24], приводит к следующему среднему значению *M* по всем возможным перестановкам элементов массива в пределе больших *N:*

$$M_{ср} \sim N(log_2 N + \gamma), \tag{2.3}$$

где γ – константа Эйлера.

Суммируя значения *C* и $M_{ср}$, получим, что среднее число операций при сортировке больших массивов методом простого выбора равно:

$$\frac{N^2}{2} + N(log_2 N + \gamma) \sim \frac{N^2}{2} + N log_2 N. \tag{2.4}$$

В соответствии с правилом построения *O*-нотации, отбрасывая члены низшего порядка и числовые коэффициенты, получаем для данного алгоритма сортировки скорость роста $O(N^2)$.

2.2.2. Сортировка включениями

Как и в предыдущем методе сортировки, массив считается состоящим из отсортированной и неотсортированной частей. Однако теперь на каждом шаге сортировки первый элемент из неотсортированной части перемещают в нужное место отсортированной части. Далее будут рассмотрены две реализации этого метода, отличающиеся способом такой «вставки».

2.2.2.1. Сортировка простыми включениями

В этом варианте алгоритма включение нового элемента в отсортированную часть осуществляется путем последовательного сдвига входящих в нее элементов вправо. Этот процесс может завершиться при выполнении одного из двух условий:

- найден элемент меньший, чем включаемый;
- достигнута левая граница массива.

Функция, реализующая алгоритм сортировки простыми включениями, представлена в листинге 2.5.

Листинг 2.5. Сортировка простыми включениями

```
public void SortInsertion(int[] a)
{
   int tmp;
   int N = a.Length;
   for (int i = 1; i < N; i++)
   {
      int j = i - 1;
      while (j >= 0 && tmp < a[j])
         a[j + 1] = a[j--]; // сдвинуть элемент
      // поставить элемент на свое место
      a[j + 1] = tmp;
   }
}
```

В сортировке простыми вставками число сравнений при перемещении очередного элемента на свое место зависит от степени упорядоченности исходного массива. Оно минимально и на каждой итерации равно 1, если исходный массив упорядочен по возрастанию. В случае упорядоченности ис-

ходного массива по убыванию на *i*-ой итерации будет выполняться *i* сравнений (`i=1,2,…,N-1`). В этой части алгоритм сортировки простыми включениями демонстрирует более естественное поведение по сравнению с алгоритмом сортировки простым выбором.

Среднее число сравнений на *i*-ой итерации равно $C_{cp}(i)=(i+1)/2$, а общее число перестановок равно:

$$C_{cp}=\sum_{i=1}^{N-1}\frac{i+1}{2}=\frac{N^2+N-2}{4}\sim\frac{N^2}{4}. \qquad (2.5)$$

Среднее число пересылок *i*-ой итерации равно $M_{cp(i)}=C_{cp(i)}+2$. Поэтому

$$M_{cp}=\frac{N^2+9N-10}{4}\sim\frac{N^2}{4}. \qquad (2.6)$$

Таким образом, скорость роста для алгоритма сортировки простыми включениями равна $O(N^2)$. Однако в сравнении с алгоритмом сортировки простым выбором имеется небольшой проигрыш в скорости, благодаря более сильной зависимости M_{cp} от N (N^2 против $N*log_2N$ в сортировке выбором). Отметим также, что этот алгоритм сортировки простыми включениями демонстрирует естественное поведение.

2.2.2.2. Сортировка бинарными включениями

Поскольку поиск места для нового элемента ведется в отсортированной части массива, то его можно ускорить за счет применения рассмотренного выше алгоритма бинарного поиска. Данная идея реализована в программе, представленной в листинге 2.6.

Листинг 2.6. Сортировка бинарными включениями

```
public void SortBinInsert(int[] a)
{
   int N = a.Length;
   for (int i=1; i<= N-1; i++)
   {
      int tmp=a[i], left=1, right=i-1;
      while (left<=right)
      {
         int m=(left+right)/ 2;
```

```
            //определение индекса среднего элемента
            if (tmp<a[m])
               right=m-1;  // сдвиг правой или
            else left=m+1; // левой границы
         }
         for (int j=i-1; j>=left; j--)
            a[j+1] = a[j]; // сдвиг элементов
   // размещение элемента в нужном месте
         a[left]=tmp;
      }
   }
```

Число сравнений $C \sim N * log_2 N$, так как поиск места вставки осуществляется только в одной части массива. Но это улучшение касается только числа сравнений. К сожалению, это улучшение не является решающим, поскольку пересылка элементов более трудоемкая операция. Поэтому число перестановок по-прежнему имеет временную сложность $\sim N^2$. Соответственно, скорость роста и для этой модификации алгоритма $O(N^2)$.

Для массивов больших размеров сортировка вставками оказывается очень неэффективным методом ввиду большого числа перемещений элементов. Очевидно, что гораздо эффективнее переставлять только некоторые элементы и на большие расстояния. Ниже будет рассмотрен метод сортировки, основанный на этой идее и известный как сортировка Шелла.

2.2.3. Сортировка обменом

Сортировка методом обмена основана на том, что многократно повторенный процесс упорядочения пар соседних элементов приводит к упорядочению всего массива.

Это обстоятельство отнюдь не является очевидным, но при некотором размышлении можно понять причину такого результата. Она заключается в том, что при проходе от начала к концу массива с упорядочением пар элементов на последнее место «выталкивается» максимальное значение. Последующие проходы не меняют его расположения, но постепенно формируют упорядоченную по возрастанию последовательность значений в конце массива.

2.2.3.1 Сортировка простым обменом

Сортировка простым обменом известна также как *метод пузырька* и в полной мере воплощает принцип обменной сортировки, описанный выше. Свое образное название этот метод получил в силу одной своей особенности:

при сортировке по возрастанию отсортированная часть массива формируется путем выталкивания («всплывания») при каждом просмотре наибольшего из значений в неотсортированной части массива. Соответственно, при сортировке по убыванию «всплывают» минимальные значения. Алгоритм сортировки простым обменом приведен в листинге 2.7.

Листинг 2.7. Сортировка методом пузырька

```
public void SortBubl(int[] a)
{
   int N = a.Length;
   for (int i = 1; i < N; i++)
     for(int j =L-1; j >= i; j--)
       if (a[j-1] > a[j])
       {
          int t = a[j-1];
          a[j-1] = a[j];
          a[j] = t;
       }
}
```

Число сравнений в этом алгоритме равно

$$C = \frac{N^2 - N}{2}. \tag{2.7}$$

Минимально число перестановок равно нулю для массива, упорядоченного по возрастанию, и совпадает с числом сравнений для массива, упорядоченного по убыванию. Поэтому

$$M_{cp} = \frac{N^2 - N}{4}. \tag{2.8}$$

Отсюда следует, что данный алгоритм обладает квадратичной скоростью роста $O(N^2)$.

2.2.3.2. Шейкер-сортировка

Предыдущий алгоритм сортировки можно попытаться усовершенствовать, используя следующие идеи:

- условием завершения процесса сортировки считать отсутствие парных перестановок при очередном просмотре;
- сравнение пар элементов производить только до места последней перестановки: раз не было перестановок, значит дальше элементы упорядочены;
- чередовать направления просмотра, что позволит одновременно формировать две отсортированных области – в левой и правой частях массива. При прохождении массива слева направо (снизу вверх) При прохождении массива слева направо (снизу вверх) поднимается легкий пузырек. При прохождении массива справа налево (сверху вниз) опускается тяжелый пузырек

Основанный на этих идеях алгоритм сортировки получил название шейкер-сортировки (от англ. *shake* – трясти). Его текст приведен в листинге 2.8.

Листинг 2.8. Шейкер-сортировка

```
public void SortShaker(int[] a)
{
   int left = 1, right = a.Length - 1, last = right;
   do
   {
      for (int j = right; j >= left; j--)
         if (a[j - 1] > a[j])
         {
            int t = a[j - 1];
            a[j - 1] = a[j];
            a[j] = t;
            last = j;
         }
      left = last;
      for (int j = left; j <= right; j++)
```

```
            if (a[j - 1] > a[j])
            {
               int t = a[j - 1];
               a[j - 1] = a[j];
               a[j] = t;
               last = j;
            }
         right = last - 1;
      }
      while (left < right);
   }
```

Все усовершенствования сортировки обменом приводят только к уменьшению числа сравнений, тогда как основное время занимает перестановка элементов. Поэтому эти усовершенствования не приводят к значительному эффекту. Анализ показывает, что сортировка методом пузырька (и даже ее улучшенный вариант – шейкер-сортировка) менее эффективна, чем сортировка вставками и обменом.

2.2.4. Сортировка Шелла

Этот метод является усовершенствованием метода вставок. Суть его заключается в разбиении сортируемого массива на ряд цепочек из равноотстоящих друг от друга элементов. Расстояние между элементами одной цепочки (шаг цепочки) первоначально выбирается достаточно большим. Элементы каждой из цепочек сортируются обычным методом вставок. Однако перестановка значений элементов, находящихся на больших расстояниях друг от друга, позволяет существенно повысить эффективность алгоритма. При последующих просмотрах шаг цепочек уменьшается до тех пор, пока он не станет равным 1.

Повышение эффективности алгоритма достигается за счет того, что на начальных этапах в сортировке участвуют немного элементов. С каждым этапом степень упорядоченности массива повышается и на последнем этапе, когда происходит сортировка простыми вставками, число перемещений элементов невелико.

В этой сортировке важен правильный выбор величины шагов. Анализ показывает, что величины шагов не должны быть кратны друг другу, чтобы достигнуть лучших результатов. Д. Кнут [24] рекомендует такие последовательности (записанные в обратном порядке):

1, 4, 13, 40, 121, … , где stepk-1=3*stepk+1,

1, 3, 7. 15, 31, …где stepk-1=2*stepk+1.

В листинге 2.9, приведенном ниже, шаги выбирались по формуле

```
stepk = stepk-1*3/5,
```
(2.9)

начиная с *step1 = N/2* и заканчивая шагом, равным единице.

Листинг 2.9. Сортировка Шелла

```
public void SortShell(int[] a)
{
   int N = a.Length;
   int step = N / 2; // первый шаг
   while (step >= 1)
   {
      int k = step;
      for (int i = k + 1; i < N; i++)
      {
         int tmp = a[i]; int j = i - k;
         while ((j > 0) && (tmp < a[j]))
         {
            a[j + k] = a[j];
            j = j - k;
         }
         a[j + k] = tmp;
      }
// определение следующего шага
      step = 3 * step / 5;
   }
}
```

Анализ алгоритма сортировки Шелла показывает, что скорость его роста *O(N)*. Это значительное улучшение по сравнению с «родительской» сортировкой простыми вставками, имеющей скорость роста $O(N^2)$.

2.2.5. Сортировка подсчетом

Все ранее рассмотренные методы обеспечивали выполнение принципа сортировки *in situ* – на том же месте. Однако при отсутствии ограничения на используемую память можно использовать в процессе сортировки дополнительные массивы. В частности, можно переписывать элементы из исходного неотсортированного массива в результирующий массив сразу в нужном порядке. Тогда число наиболее трудоемких операций – пересылок элементов – будет ровно *N*. Разумеется, для определения номера элемента в новом массиве придется сделать какое-то количество сравнений.

В качестве примера возьмем очень простую сортировку методом подсчета. Ее идея заключается в том очевидном факте, что *i*-й ключ в упорядоченном массиве превышает ровно *i*-1 остальных ключей, если никакие два ключа не равны. Таким образом, идея состоит в том, чтобы сравнить попарно все ключи и для каждого из них подсчитать количество ключей с меньшим значением. Для подсчетов числа ключей, меньших данного, используется вспомогательный массив счетчиков, значения которого служат для пересылки элементов из исходного массива в новый. Для минимального элемента исходного массива значение соответствующего элемента массива счетчиков равно нулю. Сортировка подсчетом является устойчивой.

В листинге 2.10 приведен алгоритм сортировки методом подсчета.

Листинг 2.10. Сортировка подсчетом

```
public void SortMove(int[] a)
{
   int N = a.Length;
   int[] cnt = new int[N]; // массив счетчиков
   int[] b = new int[N];// отсортированный массив
//инициализация массива счетчиков
   for (int i = 0; i < N; i++) cnt[i]=0;
//  сравнение элементов и заполнение массива счетчиков
   for (int i = 0; i < N; i++)
      for (int j = i + 1; j < N; j++)
```

```
            if (a[i] > a[j])
               ++cnt[i];
            else
               ++cnt[j];
//пересылка элементов в новый массив
   for (int i=0; i<N; i++)
      b[cnt[i]] = a[i];
   for (int i = 0; i < N; i++)
      a[i] = b[i];
}
```

Число сравнений в сортировке подсчетом C~N2, число пересылок (из исходного массива в новый) *M=N*. Поскольку операции пересылки являются более затратными по времени, чем сравнения, то скорость роста данного алгоритма близка к линейной, т.е. *O(N)*. Алгоритм дает правильный результат независимо от числа равных ключей.

2.3. Хеширование

Время выполнения алгоритмов простого и бинарного поиска по-разному, но зависит от размера массива. Идея хеширования заключается в том, чтобы эту зависимость убрать. Это достигается установлением функциональной зависимости между значением элемента массива и его индексом. В этом состоит суть так называемой *ассоциативной адресации* (*прямой адресации*, *H-кодирования*, *хеширования* – от англ. hash – мешанина, крошево). Массив, сформированный по принципу ассоциативной адресации, называется *хеш-таблицей.* Функция, устанавливающая связь между значением элемента и индексом элемента, называется *хеш-функцией.* Значение, используемое в качестве аргумента хеш-функции, будем далее называть ключом.

Таким образом, каждому ключу в хеш-таблице определено «свое» место. Поэтому при поиске какого-либо значения сразу же смотрим, есть ли оно на отведенном ему месте.

Итак, в идеальном случае предполагается, что хеш-таблица представлена с помощью массива `N` элементов, которые более удобно нумеровать от 0 до `N-1`. Кроме того, предполагается, что существует хеш-функция `h(Key)`, ставящая в соответствие каждому ключу Key некоторое целое число `i`, различное для разных значений ключа и такое, что `0<=i<=N-1`. Тогда достаточно разместить этот ключ `Key` в элементе хеш-таблицы с индексом

`i=h(Key)`, и время поиска становится постоянным. Говорят, что алгоритм поиска в хеш-таблице имеет скорость роста *O(1)*.

К сожалению, эта идеальная схема практически не работает, и вот почему. Посмотрим, каким же критериям должна удовлетворять хеш-функция. Во-первых, она должна возвращать индекс элемента в диапазоне индексов хеш-таблицы. Это легко достигается, если последним действием хеш-функции сделать вычисление остатка от деления на N.

Во-вторых, желательно иметь для каждого ключа уникальный индекс. Но этого ни одна хеш-функция гарантировать не может. Причина этого очевидна: множество возможных значений ключа либо бесконечно, либо намного превосходит размер хеш-таблицы. Поэтому в любом случае одному и тому же индексу будет соответствовать некоторое подмножество значений ключа. Например, если в качестве ключа используются строковые значения, а индекс элемента вычисляется как остаток от деления суммы числовых кодов символов строки на *N*, то два ключа `abc` и `cab` будут иметь одинаковые значения хеш-функции, то есть одинаковые индексы. Говорят, что два ключа `Key1` и `Key2`, таких, что `Key1 !=Key2` и `h(Key1)=h(Key2)`, вызывают *коллизию*.

Существует два вида коллизий: *непосредственные коллизии*, соответствующие случаю `h(Key1)=h(Key2)`, и *коллизии по модулю*, когда `h(Key1)!=h(Key2)`, но `h(Key1)=h(Key2)%N`. Долю коллизий по модулю можно уменьшить, если в качестве размера хеш-таблицы выбирать число, не имеющее делителей меньше 20. Например, это может быть простое число.

Ясно, что если таблица имеет много пустых ячеек, то коллизия разрешится быстро. Поэтому размер таблицы выбирается несколько большим, чем предполагаемое число элементов в ней. Для оценки эффективности работы с хеш-таблицей в качестве параметра используют коэффициент заполнения $\alpha=size/sizeTable$, где `size` – число элементов в таблице. Оценки [32] показывают, что среднее число сравнений для достаточно больших `size` примерно равняется $1/(1-\alpha)$ для неуспешного поиска или включения и $(1/\alpha)\log_2(1/(1-\alpha))$ – для успешного поиска. Конкретные результаты приведены в таблице 2.1.

Таблица 2.1

Среднее число обращений в хеш-таблицах

Коэффициент заполнения	**Среднее число обращений при включении или неуспешном поиске**	**Среднее число обращений при успешном поиске**
25%	1,33	1,15

50%	2	1,39
75%	4	1,85
90%	10	2,56
95%	20	3,15

Видно, что число обращений к таблице, то есть временная сложность алгоритма, зависит только от степени заполнения таблицы, а не от ее размера: в таблице, содержащей тысячи элементов, можно найти любой из них в среднем за два с половиной обращения, если только таблица заполнена не более чем на 90%.

Следует также отметить, что на эффективность поиска оказывает влияние и выбор хеш-функции. Однако в любом случае алгоритм поиска в хеш-таблице имеет скорость роста $O(1)$, т. е. не обнаруживает какой-либо зависимости от размера таблицы.

При возникновении коллизии выполняются действия, называемые *обработкой* или *разрешением* коллизии. Существуют различные методы разрешения коллизий. Рассмотрим некоторые из них.

2.3.1. Метод цепочек

Этот метод заключается в том, что с каждым элементом таблицы связывается набор значений с одинаковым значением индекса. Для хранения такого набора удобно использовать экземпляр класса `ArrayList`. В этом случае операции добавления, удаления и поиска будут разбиваться на два этапа:

- определение индекса в хеш-таблице;
- выполнение соответствующей операции в цепочке с использованием методов класса `ArrayList`.

Добавить новый элемент в конец цепочки можно с помощью метода `Add`. Поиск в цепочке элемента с заданным значением ключа выполняется методом `IndexOf`. Для удаления элемента можно воспользоваться методом `Remove`.

2.3.2. Открытая адресация

Суть метода заключается в том, что при возникновении коллизии значение полученного индекса подвергается коррекции. В простейшем случае выполняется *линейный поиск* свободного элемента таблицы. При этом значение индекса меняется по линейному закону `ind=(ind+step)%sizeTable`, где `ind` – текущее значение индекса, `step` – *шаг поиска*, `sizeTable` – размер хеш-таблицы. Величина шага поиска не должна быть делителем размера таблицы во избежание зацикливания. Недостатком данного метода является эффект группировки элементов данных в смежных эле-

ментах таблицы, что делает поиск свободного элемента более длительным и менее эффективным.

Проблема группировки устраняется при использовании *квадратичного поиска*. В этом случае шаг поиска является переменным и определяется по формуле k^2, где k – номер шага. Отметим, что квадратичный поиск гарантированно завершается успешно, если хеш-таблица заполнена меньше чем наполовину.

2.3.3. Двойное хеширование

При двойном хешировании используются две независимые хеш-функции `h1` и `h2`. Для поиска места в хеш-таблице по ключу `Key` сначала вычисляют индекс `ind0 = h1(Key)`. Если при этом возникает коллизия, то производится повторное вычисление индекса по формуле `(ind0+h2(Key))% sizeTable`, затем `(ind0+2*h2(Key))% sizeTable` и так далее. В общем случае идет проверка последовательности ячеек `(ind0+i*h2(Key))%sizeTable`, где `i=0,1,. . .,sizeTable`.

В среднем, при грамотном выборе хеш-функций двойное хеширование обеспечивает меньшее число попыток по сравнению с линейным поиском за счет того, что вероятность совпадения значений сразу двух независимых хеш-функций ниже, чем одной.

2.3.4. Проект «Телефонный справочник»

Рассмотрим работу с хеш-таблицей на примере простейшего телефонного справочника. В каждом элементе хеш-таблицы будем хранить структуру, содержащую два строковых поля: номер телефона и фамилию абонента.

```
struct TInfo
{
   public string phone;
   public string fio;
}
```

В качестве ключа будем использовать номер телефона. Тогда можно предложить хеш-функцию, код которой представлен в листинге 2.11.

Листинг 2.11. Хеш-функция

```
int hashKey(string s)
{
   int result = 0;
```

```
        for (int i = 0; i < s.Length; i++)
        {
            result += Convert.ToInt32(s[i]) * i;
            result %= sizeTable;
        }
        return result;
    }
```

Выбор хеш-функции является самой сложной проблемой хеширования. В этой функции с целью исключения коллизии для телефонных номеров (например, 123456 и 654321) суммируются коды цифр с весами, равными порядковому номеру цифры.

Операции с хеш-таблицей:

- добавление элемента в таблицу;
- удаление элемента из таблицы;
- поиск элемента в таблице.

При выполнении этих операций могут возникать коллизии и поэтому необходимо предусмотреть метод их разрешения. Будем использовать метод открытой адресации с линейным поиском. Из дальнейшего станет ясно, что для реализации функции поиска необходимо наличие у элемента хеш-таблицы признаков посещаемости и занятости. Таким образом, структура элемента хеш-таблицы должна иметь следующий вид:

Листинг 2.12. Структура элемента в хеш-таблице

```
struct THashItem
{
    public TInfo info;
    public bool empty;  // признак занятости
    public bool visit;  // признак посещения
}
```

Создадим класс для работы с хеш-таблицей.

Листинг 2.13. Класс хеш-таблицы

```
class MyHash
```

```
{
   public int sizeTable; // размер таблицы;
   static int step = 37; //шаг для разрешения
//коллизий
   int size;            // число элементов в таблице
   public THashItem[] h; // хеш-таблица
   public MyHash(int sizeTable) // конструктор
   public void HashInit()   // метод инициализации
   int hashKey(string s)    // хеш-функция
   public int AddHash(string fio, //метод добавления
//элемента
     string phone)
   void ClearVisit()    //метод очистки полей
//посещения
   public bool DelHash(string phone, out int i)
//метод удаления элемента
   public int FindHash(string phone, // метод поиска
//по ключу
      out string FIO, out int count)
}
```

Конструктор класса задает размер таблицы и инициализирует массив *h*[].

Листинг 2.14. Конструктор класса

```
public MyHash(int sizeTable)
{
   this.sizeTable = sizeTable;
   h = new THashItem[sizeTable];
   HashInit();
}
```

Метод *HashInit()* представлен в листинге 2.15.

Листинг 2.15. Метод инициализации

```
public void HashInit()
{
   size = 0;
   for (int i=0; i<sizeTable; i++)
   {
      this.h[i].empty = true;
      this.h[i].visit = false;
   }
```

Для демонстрации работы с хеш-таблицей создадим оконное приложение. На форме разместим элемент управления *DataGridView*, два элемента *TextBox* для ввода и вывода данных, а также кнопки для выполнения операций заполнения таблицы, добавления, удаления и поиска абонентов (рис. 2.1).

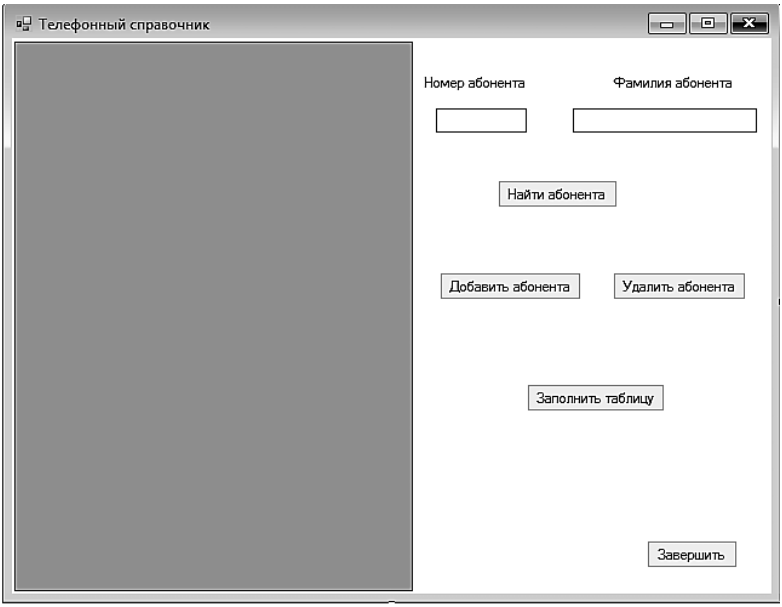


Рис. 2.1. Форма с элементами управления

Создание объекта происходит в классе *FormMain*, там же подготавливается элемент *dataGridView1*.

Листинг 2.16. Подготовка элементов в FormMain()

```
MyHash myHash = new MyHash(301);


public Form1()
{
   InitializeComponent();
   dataGridView1.Columns.Add("Номер", "Номер");
   dataGridView1.Columns["Номер"].Width = 50;
   dataGridView1.Columns.Add("Телефон", "Телефон");
   dataGridView1.Columns["Телефон"].Width = 80;
   dataGridView1.Columns.Add("Фамилия", "Фамилия");
   dataGridView1.Columns["Фамилия"].Width = 140;
   dataGridView1.AllowUserToAddRows = false;
   btnAdd.Visible = false;
   btnDel.Visible = false;
}
```

Для заполнения хеш-таблицы используются данные из текстового файла, состоящего из пар строк, первая из которых содержит фамилию абонента, вторая – номер телефона. Метод, реализующий процедуру заполнения, представлен в листинге, происходит при нажатии клавиши *buttonAdd180*:

Листинг 2.17. Заполнение хеш-таблицы

```
void FillHashTable()
{
   FileStream file = new FileStream("phoneBook.txt",
FileMode.Open);
   StreamReader strReader = new StreamReader(file);
   dataGridView1.RowCount = myHash.sizeTable;
   string fio, phone;
   for (int i = 0; i < myHash.sizeTable; i++)
```

```
        dataGridView1[0, i].Value = Con-
vert.ToString(i);
    while ((fio = strReader.ReadLine()) != null &&
              (phone = strReader.ReadLine()) != null)
        myHash.AddHash(fio, phone);
    strReader.Close();
}
```

Этот метод вызывается из обработчика клика для кнопки «Заполнить таблицу».

Листинг 2.18. Обработчик клика по кнопке заполнения

```
private void buttonAdd180_Click(object sender, Even-
tArgs e)
{
    FillHashTable();
    ShowHash();
    buttonAdd180.Visible = false;
    btnAdd.Visible = true;
    btnDel.Visible = true;
}
```

При добавлении элемента в хеш-таблицу может возникнуть ее переполнение. Поэтому алгоритм добавления оформлен в виде функции целого типа: результатом ее будет индекс добавленного элемента или –1, если таблица переполнена. Алгоритм добавления элемента в хеш-таблицу приведен в листинге 2.19.

Листинг 2.19. Добавление элемента в хеш-таблицу

```
public int AddHash(string fio, string phone)
{
    int adr = -1;
    if (size < sizeTable)
    {
      adr = hashKey(phone);    // таблица не перепол-
нена
```

```
            while (!h[adr].empty)
               adr = (adr + step) % sizeTable;
                  // место свободно - можно ставить эле-
             мент
            h[adr].empty = false;  //признак занятости
            h[adr].visit = true;   //признак посещенности
            h[adr].info.fio = fio;
            h[adr].info.phone = phone;
            size++;      //количество элементов увеличилось
         }
         return adr;
      }
```

Для поиска элемента в хеш-таблице используется метод *FindHash*, представленный в листинге 2.20.

Листинг 2.20. Поиск элемента в хеш-таблице

```
      public int FindHash(string phone,
                  out string fio, out int count)   // по-
      иск в хеш-таблице
      {
         int result = -1;
         bool ok;
         fio = "";
         count = 1;
         ClearVisit();
         int i = hashKey(phone);
         ok = h[i].info.phone == phone;
         while (!ok && !h[i].visit)
         {
            count++;
            h[i].visit = true;
            i = (i + step) % sizeTable; // продолжаем по-
      иск
            ok = h[i].info.phone == phone;
         }
```

```
    if (ok)
    {
       result = i;
       fio = h[result].info.fio;
    }
    return result;
}
```

Если элемент найден, то значение функции равно индексу, иначе –1. Дополнительная информация о найденном элементе передается параметром *fio*, параметр *count* возвращает число шагов.

Функция удаления элемента – булевская. Если элемент был найден в таблице и, следовательно, удален, то ее значение равно *true*, если элемента в таблице не было – не было и удаления, значение функции – *false*. Алгоритм приведен в листинге 2.21.

Листинг 2.21. Удаление элемента из хеш-таблицы

```
public bool DelHash(string phone, out int i)
{
   bool result = false;
   i = 0;
   if (size != 0) // таблица не пуста
   {
     int i = hashKey(phone);
     // вычисление индекса элемента
     if (h[i].info.phone == phone)
     // элемент найден
     {
        h[i].empty = true;
        result = true;
        size--;
     }
     else  // коллизия
     {
        string FIO; int count;
```

```
            i = FindHash(phone, out FIO, out count);
            if (i != -1)
            // Элемент найден. Можно удалять
            {
                h[i].empty = true;
                result = true;
                size--;
             }
          }
        }
        return result;
    }
```

Если элемент удален, то значение функции равно *true*. Если элемента в таблице не было, то значение функции — *false*.

Работающее приложение представлено на рисунке 2.2.

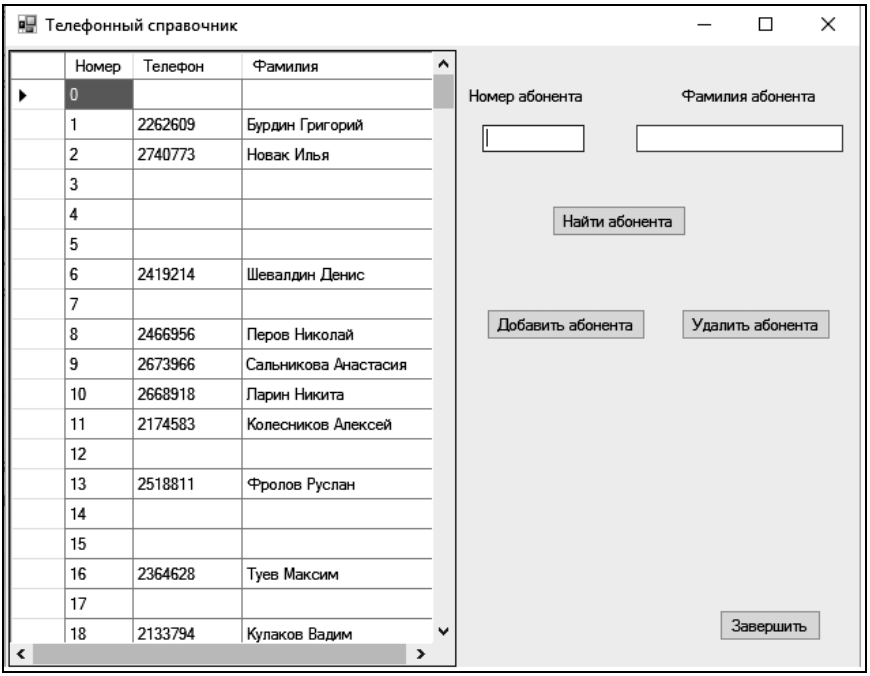


Рис. 2.2. Заполненная хеш-таблица

2.3.5. Класс Hashtable

Для работы с хеш-таблицами пользователю совсем необязательно создавать собственные классы. Платформа *.Net Framework* предоставляет готовое решение – класс *Hashtable*, определенный в пространстве имен *System.Collections*. Этот класс относится к категории *коллекций-словарей*, т.е. коллекций, элементами которых являются пары *ключ-значение*. Как ключ, так и значение могут быть данными любого типа – целыми числами, строками, объектами и т. д. Самыми важными качествами хорошего словаря являются простота добавления новых записей и быстрота получения значений. Некоторые словари организованы так, что записи добавляются в них очень быстро; другие настроены на быстрое извлечение информации. Одним из примеров словаря является хеш-таблица.

Класс *Hashtable* имеет встроенную хеш-функцию, обращение к которой происходит через вызов метода *GetHashCode*(*Key*). Для разрешения коллизий используется метод цепочек. Для хеш-таблицы класса *Hashtable* введено понятие *коэффициента загрузки* (*load factor*), который определяет максимальное значение отношения количества хранящихся в ней элементов данных к длине таблицы. При этом следует иметь в виду, что с каждым занятым элементом таблицы связана цепочка, содержащая в общем случае несколько элементов данных. Чем меньше коэффициент загрузки, тем выше производительность операций в хеш-таблице, но тем больше расходуется памяти. По умолчанию коэффициент загрузки равен 1.0, что, по утверждению фирмы Microsoft, обеспечивает оптимальный баланс между скоростью и размером. Однако программист, создающий экземпляр класса *Hashtable*, может задать любой коэффициент загрузки.

По мере добавления новых записей в объект *Hashtable* фактическая загрузка увеличивается, пока она не сравняется с коэффициентом, указанным во время создания объекта. Достигнув заданного коэффициента, объект *Hashtable* автоматически увеличивает длину таблицы до наименьшего простого числа, большего, чем удвоенная текущая длина.

Конструктор класса имеет несколько перегрузок, позволяющих, в частности, создавать хеш-таблицы с размером по умолчанию, с указанным размером, с указанными размером и коэффициентом загрузки и т. д. Класс *Hashtable* имеет свойство-индексатор, что позволяет использовать для его экземпляров нотацию массивов с ключами в качестве индексов. В таблице 2.2 представлены основные свойства и методы класса *Hashtable*.

Таблица 2.2

Основные свойства и методы класса *Hashtable*

Метод или свойство	Описание
Count	Открытое свойство, позволяющее узнать текущее число элементов
Keys	Открытое свойство, возвращающее коллекцию ключей
Values	Открытое свойство, возвращающее коллекцию значений
Add()	Добавляет запись с указанной парой ключ-значение
Clear()	Удаляет все элементы из объекта Hashtable
ContainsKey()	Выясняет, содержит ли объект Hashtable указанный ключ
ContainsValue()	Выясняет, имеется ли в объекте Hashtable указанное значение
GetHashCode()	Возвращает индекс в хеш-таблице по заданному ключу
Remove()	Удаляет запись с указанным ключом

В листинге 2.22 приведен пример использования объекта класса *Hashtable*. Отметим, что в данном случае использование метода *ToString()* при выводе результата является избыточным и сохранено для универсальности кода.

Листинг 2.22. Пример использования объекта класса Hashtable

```
static void Main(string[] args)
{
   Hashtable hash = new Hashtable(29);
   hash.Add("Имя, фамилия", "Андрей Капустин");
   hash.Add("Возраст", 30);
   hash.Add("Специальность", "IT-специалист");
   hash.Add("Зарплата", 60000);
```

```
    hash["Пол"] = "мужской";
    foreach (Object key in hash.Keys)
        Console.WriteLine(key.ToString() + ":\t" +
hash[key].ToString());
    Console.ReadLine();
}
```

Результат выполнения этого кода показан на рисунке 2.3. Как и ожидалось, последовательность, в которой происходит вывод данных, не совпадает с последовательностью их ввода, что обусловлено особенностями размещения данных в хеш-таблице.

```
Пол:        мужской
Зарплата: 60000
Имя, фамилия:    Андрей Капустин
Специальность:   IT-специалист
Возраст:  30
```

Рис. 2.3. Результат выполнения кода листинга 2.22

Выводы

Алгоритмы поиска и сортировки являются фундаментальными алгоритмами информатики. Скорость их выполнения очень часто оказывает решающее влияние на скорость работы различных программных систем. Именно поэтому их разработке и анализу уделяется столь пристальное влияние. Одним из наиболее полных является обзор методов сортировки и поиска, представленный в монографии [24]. В данной главе мы ограничились рассмотрением лишь наиболее распространенных из них. Среди рассмотренных были алгоритмы с различной временн***о***й эффективностью: O(1) для поиска в хеш-таблице, $O(\log_2 N)$ для бинарного поиска, O(N) для сортировки Шелла и сортировки подсчетами, $O(N \cdot \log_2 N)$ для сортировки выбором, $O(N_2)$ для сортировок включениями и обменом. Выбор того или иного алгоритма должен определяться выбором между требованиями к скорости выполнения программы и сложностью реализации алгоритма.

Для повышения эффективности операций поиска в больших массивах данных рекомедуется использовать технологию хеширования, позволяющую за счет некоторого избыточного расходования ресурса памяти существенно повысить скорость поиска. Для работы с хеш-таблицами рекомендуется использовать готовое решение Microsoft – класс *`Hashtable`*.

Упражнения

1. Используя класс `Timing` (глава 1), проведите экспериментальное изучение зависимости времени выполнения от размера массива для алгоритмов сортировки методом включения и методом Шелла.
2. То же для алгоритмов простого и бинарного поиска.
3. Каков будет результат выполнения алгоритма бинарного поиска для неупорядоченного массива?
4. Создайте приложение, в котором подсчитывается количество сравнений при выполнении поиска в массиве. Подсчитайте количество сравнений при использовании алгоритмов простого поиска, поиска с барьером и бинарного поиска для одних и тех же исходных данных.
5. В приложении «Телефонный справочник» реализуйте разрешение коллизий методом цепочек.
6. Используя класс `Hashtable`, создайте справочник столиц стран мира. В качестве ключа используйте название страны, а в качестве значения – название ее столицы.

Глава 3. Рекурсия

Рекурсия относится к числу фундаментальных понятий математики и информатики. В математике *рекурсивной* называют функцию, в определении которой содержится сама эта функция. Часто это функции целочисленного аргумента, которые задаются с помощью *рекуррентных отношений.* Примерами таких отношений являются определения функции факториала:

$$n! = n(n-1)!,\ 0! = 1$$

и числовой последовательности Фибоначчи:

$$F(n) = F(n-1) + F(n-2),\ F(0) = F(1) = 1.$$

В математике широко используются рекурсивные определения математических объектов. В качестве примера можно привести рекурсивное определение натурального числа:

- 1 есть натуральное число;
- целое число, следующее за натуральным, есть натуральное число.

3.1. Рекурсивные определения и рекурсивные алгоритмы

Мощность рекурсии определяется тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания. В частности рекурсивные определения широко используются для описания языков программирования. Например, в языке C# оператор — это и заключенная в операторные скобки { и } последовательность операторов, и оператор цикла, телом которого также является оператор. Благодаря рекурсивным определениям и существует все бесконечное разнообразие программ.

В программировании рекурсивными могут быть как структуры данных, так и алгоритмы. Характерной рекурсивной структурой являются деревья, которые будут рассматриваться в следующей главе.

Бесконечные вычисления можно описать с помощью конечной рекурсивной программы, даже если эта программа не содержит явных циклов. Необходимое и достаточное средство для рекурсивного задания алгоритмов – это представление их в виде функций, содержащих в своем теле вызовы самих себя. Это называется *прямой рекурсией*. Если функция *P* содержит обращение к функции *Q*, которая содержит обращение к *P*, то функция *P* называется *косвенно рекурсивной*.

Любая функция, вообще говоря, содержит некоторое множество локальных объектов, т.е. объектов, которые определены в самой этой функции. Когда функция вызывает сама себя, то для всех ее локальных переменных выделяется новая память в системном стеке, и вложенный вызов работает с собственным представлением локальных переменных. Когда вложенный вызов завершается, занимаемая его переменными область памяти в стеке освобождается и актуальным становится представление локальных переменных предыдущего уровня. Хотя локальные переменные имеют те же имена, что и соответствующие переменные, созданные при предыдущем вызове этой же процедуры, их значения различны. Это обусловлено тем, что при работе процедуры используются переменные, созданные последними. Это же относится и к параметрам функции.

Рекурсивные функции могут привести к бесконечным вычислениям. При написании рекурсивной программы следует позаботиться о том, чтобы ее работа когда-либо завершилась. Для этого необходимо, чтобы рекурсивное обращение к функции *P* подчинялось некоторому условию *C*, которое в какой-то момент перестанет выполняться. Завершение рекурсии хорошо видно в определении факториала: при $n = 1$ факториал вычисляется *непосредственно*.

Следовательно, рекурсивная функция *P* должна иметь вид:

```
if (C)
  P;        // рекурсивный вызов
else T;     // непосредственное вычисление или дейст-
вие
```

или

```
while (C)
  P;        // рекурсивный вызов
T;          // непосредственное вычисление или дейст-
вие
```

или некоторую другую эквивалентную форму.

В ряде случаев завершением выполнения соответствующей функции может управлять некоторая связанная с этой функцией величина *N*, то есть целое неотрицательное число, относительно которого можно было бы с уверенностью сказать, что оно убывает при каждом рекурсивном вызове функции. В простейшем случае роль такой управляющей величины может играть один из параметров функции. И тогда любая функция, построенная по следующей схеме:

```
void P(int N,  …);
{
```

```
  if (N == 0)
    T;              // непосредственное вычисление или
действие
  else
    P(N-1, …);      // рекурсивный вызов с параметром,
                    // на единицу меньшим
}
```

после конечного числа рекурсивных вызовов выдаст определенный результат для каждого неотрицательного *N*.

В качестве примера рассмотрим функцию, которая читает последовательность символов, завершающуюся точкой, и выводит ее в инвертированном виде.

Листинг 3.1. Печать в обратном порядке

```
static void Invert()
{
   Char ch = Convert.ToChar(Console.Read());
   if (ch != '.')
      Invert();
   Console.Write(ch);
}
```

Работа функции *Invert()* для входной последовательности '*a*' '*b*' '*c*' '.' показана на рисунке 3.1.

При каждом вызове функции *Invert*() выделяется место под локальную переменную *ch*, значения которой приведены в правом верхнем углу программного текста. Стрелками с цифрами в кружках обозначены рекурсивные вызовы (1, 2, 3). Стрелка (4) показывает работу функции в том случае, когда условие рекурсивного обращения к функции не выполняется. Возвраты к предыдущему вызову обозначены стрелками (5, 6, 7). Стрелка (8) обозначает возврат в вызывающую программу при полном завершении работы функции.

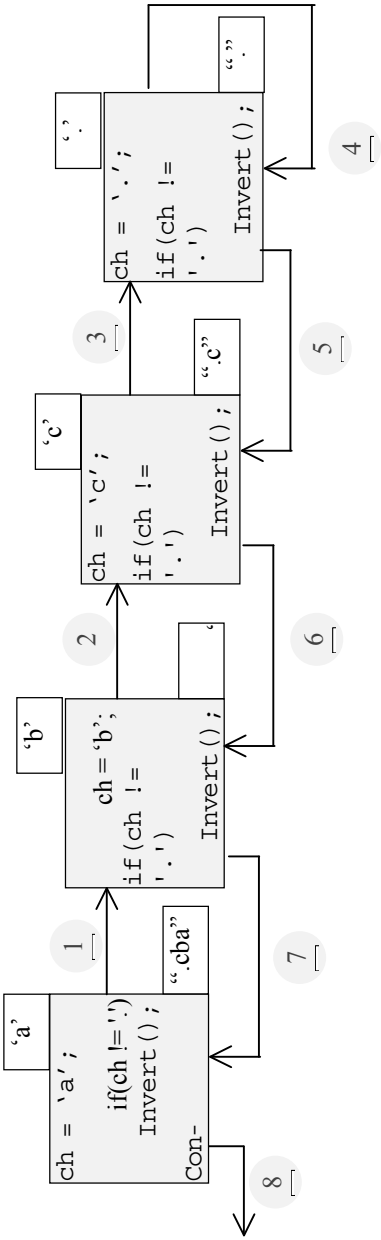


Рис. 3.1. Последовательность рекурсивных вызовов процедуры *Invert()*

3.2. Когда рекурсия необходима

Далеко не всегда рекурсия бывает необходимой. Напомним, что при каждом рекурсивном вызове функции выделяется память для размещения ее переменных. Кроме этих локальных переменных нужно еще сохранять текущее состояние вычислений, чтобы вернуться к нему, когда закончится выполнение новой активации функции и нужно будет вернуться к старой. Все эти данные, а также и адрес возврата операционная система помещает в системный стек. Эти накладные расходы необходимо учитывать при выборе между рекурсивным и нерекурсивным решением.

Для примера рассмотрим рекурсивную функцию вычисления факториала. Определение факториала по своей сути является рекурсивным, и, казалось бы, алгоритм его вычисления тоже должен быть таким же. Приведенная ниже функция `Factorial` является калькой определения:

Листинг 3.2. Вычисление факториала. Вариант 1

```
static int Factorial(int k)
{
   int result;
   if (k == 1)
      result = 1;
   else
      result = k * Factorial(k-1);
   return result;
 }
```

Однако итеративный алгоритм вычисления факториала нисколько не сложнее, и к тому же лишен накладных расходов, связанных с рекурсией:

Листинг 3.3. Вычисление факториала. Вариант 2

```
static int FactorialI(int k)
{
   int result = 1;
   for (int i = 1; i <= k; i++)
      result *= i;
   return result;
}
```

На ряде других аналогичных примеров можно показать, что если имеется *очевидное* итеративное решение задачи, то следует избегать рекурсии.

Любую рекурсивную программу можно преобразовать в чисто итеративную. Но для этого придется самим создавать стек для рекурсивных вызовов и оперировать с ним, и порой эти операции до такой степени заслоняют основной алгоритм, что понять его становится весьма и весьма трудно. Поэтому алгоритмы, которые по своей природе скорее рекурсивны, чем итеративны, следует представлять в виде рекурсивных процедур.

3.3. Примеры рекурсивных программ

Методика создания рекурсивных программ в общем случае состоит из таких этапов:

- *параметризация задачи*, то есть выделение тех элементов, от которых зависит решение и в частности размерность задачи; размерность должна убывать при каждом рекурсивном вызове;
- поиск *тривиального случая,* который может быть разрешен без рекурсивного вызова, и его решение.

Последний этап является ключевым при построении рекурсивного алгоритма и ему обычно соответствует размерность задачи, равная нулю или единице.

Далее приведены два примера рекурсивных программ – «Ханойские башни» и сортировка массива.

3.3.1. Задача о Ханойских башнях

Легенда о Ханойских башнях была придумана математиком Эдуардом Люка в начале XIX века. Она представляла собой увлекательную формулировку математической задачи, и вызвала такой большой интерес, что множество людей старались найти ее решение. А задача и связанная с нею легенда такова.

У служителей одного из монастырей, расположенного в труднодоступных горах, есть три алмазных стержня. На одном из стержней лежат 64 золотых кольца, различных по размеру и расположенных пирамидой: снизу самого большого диаметра, потом меньшего и, наконец, наверху пирамиды — самый маленький. Эти кольца нужно переложить на другой стержень, получив точно такую же пирамиду. Третий стержень используется как промежуточный. Условия перемещения колец следующие:

- за один раз можно перемещать только одно кольцо;
- кольцо можно брать только с вершины одной из пирамид, а класть либо на пустой стержень, либо на кольцо большего диаметра.

Когда монахи перенесут все 64 кольца, наступит конец света. На рисунке 3.2 изображены три стержня, на одном из которых находится пирамида из 4-х колец.

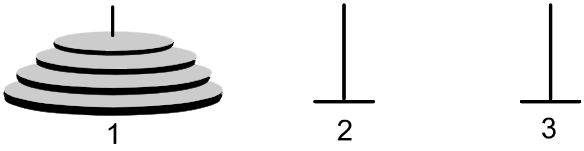


Рис. 3.2. Ханойские башни (начальное положение)

Поставим задачу вывода на экран последовательности перемещений колец. Данная задача имеет очевидное рекурсивное решение.

Напомним, что рекурсивное решение любой задачи состоит из этапов параметризации и поиска тривиального случая. Здесь естественным параметром является *K* – число колец. Кроме того, в число параметров нужно включить: номер стержня, с которого надо снять пирамиду колец, номер промежуточного стержня и номер стержня, на который пирамиду надо перенести. Тривиальным будет случай, соответствующий *K* = 0, не требующий выполнения каких-либо действий.

K колец могут быть перемещены со стержня 1 на стержень 2 путем:

- рекурсивного переноса *K–1* колец со стержня 1 на стержень 3 с учетом правил игры. Стержень 2 используется как промежуточный;
- перемещения на стержень 2 со стержня 1 последнего — наибольшего — кольца (рис. 3.3);
- рекурсивного переноса *K–1* колец со стержня 3 на стержень 2. Но теперь как промежуточный будет использоваться стержень 1.

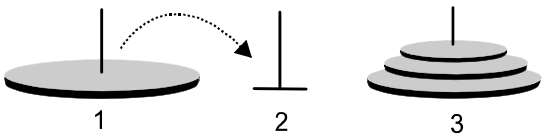


Рис. 3.3. Ханойские башни (промежуточное положение)

С учетом сказанного структура функции `Hanoj()` будет выглядеть так, как это представлено в листинге 3.4.

Листинг 3.4. Ханойские башни

```
void Hanoj(int d1,int d2,int d3, // стержни
           int k)                // количество колец
```

```
{
  if (k>0)
  {
    Hanoj(d1,d3,d2,k-1);
    //переместить кольцо со стержня d1 на стержень
d2
    //отобразить
    Hanoj(d3,d2,d1,k-1);
  }
}
```

В этой функции пока не прописаны два действия:

- переместить кольцо со стержня *d1* на стержень *d2*,
- отобразить.

Конкретный вид этих действий зависит от способа представления данных, в частности результата выполнения функции. Мы ставим своей целью получить графическую иллюстрацию выполнения алгоритма.

Для рисования башен необходимо знать, сколько колец и какого размера лежит на каждом стержне. Поскольку по правилам игры брать со стержня можно только верхнее кольцо, класть кольцо можно только на верхушку стержня, то естественно в качестве структуры для хранения информации о кольцах на стержнях выбрать стек. В соответствии с условием задачи необходимо иметь три одинаковых стека. Мы реализуем это в виде массива стеков *ArSt*. Каждый из стеков представляет собой экземпляр класса *MyStack*, имеющего методы добавления и извлечения элементов. Данный класс описан в главе 8 книги «C#. Введение в программирование».

В стек помещаются целые числа, величина числа соответствует размеру кольца. Каждый стержень имеет свой стек. И тогда перемещение кольца с одного стержня на другой эквивалентно извлечению числа из одного стека и помещению его в другой стек. После каждого перемещения кольца отображаются башни. Функция *Hanoj()* с детализацией действия по перемещению кольца приведена в листинге 3.5.

Листинг 3.5. Ханойские башни. Основная функция

```
void Hanoj(int d1,int d2,int d3, // стержни
                int k)           // количество колец
```

```
{
    if (k>0)
    {
        Hanoj(d1,d3,d2,k-1);
        int e = ArSt[d1].PopStack();// кольцо диаметром e берется со стержня d1
        ArSt[d2].PushStack(e);      // кольцо кладется на стержень d2
        Drawing();                  // отображение стержней
        Hanoj(d3,d2,d1,k-1);
    }
}
```

Создадим проект, иллюстрирующий работу по перекладыванию колец. Форма проекта показана на рисунке 3.4. На ней представлен завершающий этап работы: 5 колец со стержня 1 перемещены на стержень 2. На форме размещены следующие элементы управления:

- текстовое поле для ввода числа колец;
- кнопка *buttonRun*, активизирующая процедуру *Hanoj*();
- кнопка *buttonClear*, удаляющая старый рисунок.

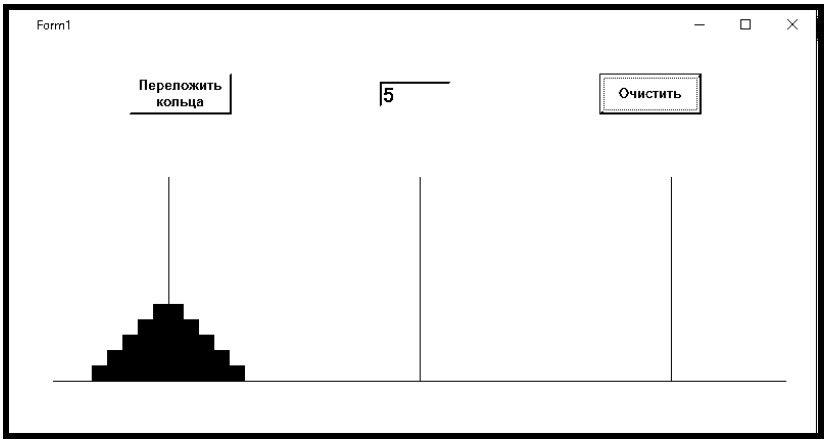


Рис. 3.4. Форма проекта «Ханойские башни»

Обработчик события щелчка по кнопке *buttonRun* приведен в листинге 3.6.

Листинг 3.6. Ханойские башни. Обработчик события `Click` кнопки `Run`

```
private void buttonRun_Click(object sender,
    EventArgs e)
{
   int n = 4;
   for (int i = 0; i < 3; i++)
     ArSt[i] = new MyStack(); //инициализация стеков
   for (int i = n; i > 0; i--)
      ArSt[0].PushStack(i); //заполнили стек, соответствующий 1-му стержню
   Drawing();
   Hanoj(0, 1, 2, n);
}
```

Функция рисования башен представлена в листинге 3.7.

Листинг 3.7. Рисование башен

```
void Drawing()  {
  int[] x0 ={80,202,324};
  int y0  = 190;   // y max
  int yH  = 40;    // y min
  int w   = 15;    // полуширина минимального кольца
  int h   = 13;    // высота кольца
  int e;
  gBitmap.Clear(Color.White);
  gBitmap.DrawLine(MyPen0, x0[0], y0, x0[0], yH);
  gBitmap.DrawLine(MyPen0, x0[1], y0, x0[1], yH);
```

```
        gBitmap.DrawLine(MyPen0, x0[2], y0, x0[2], yH);
        for (int i=0; i <= 2; i++)
        {
           MyStack tmp = new MyStack();
           while (!ArSt[i].StackIsEmpty())
           {
              e = ArSt[i].PopStack();
              tmp.PushStack(e);
           }
           int y = y0;
           while (!tmp.StackIsEmpty())
           {
             e = tmp.PopStack(); n
             ArSt[i].PushStack(e);
             gBitmap.FillEllipse(MySolidBrush,x0[i]-e*w,y,
                2*e*w, h);
             y -= h;
           }
        }
        gScreen.DrawImage(bitmap, ClientRectangle);
        Thread.Sleep(500);
    }
```

А как же «конец света»? Время его наступления зависит от числа *M(n)* перемещений колец, выполняемых функцией *Hanoj()*. Из структуры функции следует, что

$$M(n) = 2 \cdot M(n-1) + 1, \quad M(0) = 0. \tag{3.1}$$

Решение этого уравнения имеет вид:

$$M(n) = 2^n - 1. \tag{3.2}$$

Данный алгоритм имеет экспоненциальную скорость роста $O(2^n)$. Время его выполнения резко возрастает с ростом n и становится весьма значительным уже при $n > 10$.

Ну а что касается легенды, то при числе колец 64 число перемещений равно $2^{64} - 1$ или примерно 10^{20}. Если монахи переносят одно кольцо в секунду, то им потребуется свыше пяти миллиардов столетий. Так что время у нас еще есть!

3.3.2. Быстрая сортировка

Быстрая сортировка основана на сортировке обменом, которая среди всех сортировок является наименее эффективной: классический пример – сортировка методом пузырька. Однако оказывается, что усовершенствование сортировки, основанной на обмене, дает лучший из всех известных методов сортировки массивов. Этот метод изобрел К. Хоор и дал ему название *быстрая сортировка*. Хотя этот метод можно было бы назвать *сортировкой сегментацией*.

Идея этого метода основана на том факте, что наибольшая эффективность достигается, если производить обмены элементов, отстоящих друг от друга на большом расстоянии. Так, например, если исходный массив упорядочен по убыванию, то всего за `N/2` обменов его можно отсортировать: поменять местами первый и последний элементы, второй и предпоследний и так далее, продвигаясь от концов массива к его середине. Этот экзотический пример приводит к рассмотрению следующего алгоритма.

Выберем некоторым образом элемент x и будем просматривать массив, двигаясь слева направо, пока не найдем элемент $a[i]>x$, затем будем просматривать массив справа налево, пока не найдем элемент $a[j]<x$. Поменяем местами эти элементы и продолжим «просмотр с обменом», пока индексы i и j не встретятся где-то в середине массива. В результате массив разделится на две части: в левой значения элементов будут меньше, чем x, а в правой — больше x. Это пока не сортировка — только разделение. Для полной сортировки необходимо то же самое проделать с левой и правой частями массива, затем с частями этих частей, и так до тех пор, пока каждая часть не будет содержать только один элемент.

Мы получили типичный рекурсивный подход:

- параметризация — обрабатывается подмассив $a[i, j]$, для всего массива $i = 1$, $j = N$;
- тривиальный случай $i = j$, когда не выполняется никаких действий.

В качестве x будем брать значение среднего элемента. Оказывается, что такой выбор в подавляющем большинстве случаев распределения элементов в исходном массиве приводит к хорошим результатам. Более подробно о выборе x можно прочитать в [32]. Реализация функции быстрой сортировки приведена в листинге 3.8.

Листинг 3.8. Быстрая сортировка. Рекурсивный вариант

```
void QuickSort(int left, int right)
{
   int i = left; int j = right;
   int x = arr[(left + right) / 2]; // средний
элемент
   do
   {
     while (arr[i] < x)
     // поиск элемента больше среднего
       i++;
     while (arr[j] > x)
     // поиск элемента меньше среднего
       j--;
     if (i <= j)   // обмен элементов местами
     {
        int tmp = arr[i]; arr[i] = arr[j];
        arr[j] = tmp;
        i++; j--;
     }
   }
   while (i<=j);
   if (left<j)
      QuickSort(left,j);
      //обработка левого подмассива
   if (i<right)
      QuickSort(i,right);
      // обработка правого подмассива
}
```

Алгоритм быстрой сортировки имеет минимальную сложность порядка $N \times logN$ и максимальную сложность порядка N^2, но такая сложность встречается достаточно редко.

Сейчас покажем, как можно представить алгоритм быстрой сортировки в виде нерекурсивной функции. При этом рекурсия представляется как итерация, но необходимы дополнительные переменные и действия для хранения информации: отложенных вызовов. Рекурсия использует стек в скрытом от программиста виде, но при нерекурсивном представлении функции стек приходится использовать явно.

Один этап быстрой сортировки разбивает исходный массив на два подмассива. Границы правого подмассива запоминаются в стеке, и происходит сортировка левого подмассива. Затем из стека выбираются границы, находящиеся в вершине, и обрабатывается подмассив, определяемый этими границами. В процессе его обработки в стек может быть записана новая пара границ и т. д. При начальных установках в стек заносятся границы исходного массива. Сортировка заканчивается, когда стек становится пустым.

Поскольку в стеке нужно хранить левую и правую границы, тип элемента стека должен быть следующим:

```
structur TInf
{
    int l,r;
}
```

Так же, как и в задаче с Ханойскими башнями, будем использовать экземпляр класса *MyStack*. Функция *QuickStack*, реализующая нерекурсивный вариант быстрой сортировки, представлена в листинге 3.9.

Листинг 3.9. Быстрая сортировка. Нерекурсивный вариант

```
void QuickStack()
{
   int left = 0; int right = arr.Length - 1;
   MyStack stack = new MyStack();
   TInf LRRange = new TInf(left, right);
   stack.PushStack(LRRange);
   do  {
      LRRange = stack.PopStack();
      left = LRRange.l; right = LRRange.r;
```

```
do  {
   int i = left; int j = right;
   int x = arr[(left+right)/2];
   do  {
      while (arr[i] < x) i++;
      while (arr[j] > x) j--;
      if (i <= j)  {
         int tmp = arr[i]; arr[i] = arr[j];
         arr[j] = tmp;
         i++; j--;
      }
  }
  while (i<=j);
  if (j - left < right - i)  {
    if (i < right)  {
       LRRange.l = i; LRRange.r = right;
       stack.PushStack(LRRange);
    }
    right = j;
  }
  else   {
    if (left < j)  {
       LRRange.l = left; LRRange.r = j;
       stack.PushStack(LRRange);
    }
    left = i;
  }
}
while (left < right);
```

```
        }
        while (!stack.StackIsEmpty());
    }
```

Алгоритм быстрой сортировки следует применять для больших (*N>15*) массивов, так как для массивов меньшего размера он становится неэффективным. Неэффективен этот алгоритм и на частично упорядоченных массивах. Поэтому рекомендуется прерывать алгоритм быстрой сортировки, когда размер обрабатываемого подмассива достигнет порогового значения, и продолжить сортировку каким-либо другим методом, например, вставками, более эффективным для частично упорядоченного массива.

3.4. Алгоритмы с возвратом

Поиском с возвратом или бэктрекингом (от англ. *backtrack* – отходить, отступать) называется метод решения задач, сводящихся к перебору элементов некоторого множества. При этом элементы множества не являются изначально заданными, но конструируются в процессе решения. Классическими примерами таких задач являются задачи поиска пути в лабиринте, задача коммивояжера, шахматные задачи. Целью перебора может быть нахождение элемента множества, обладающего определенными свойствами (пути, ведущего к выходу из лабиринта, или последовательности ходов, обеспечивающих выигрыш в некоторой шахматной позиции). Но может стоять задача перебора всех возможных элементов множества, например, всех путей между двумя вершинами графа. Подобные задачи особенно часто встречаются в области *искусственного интеллекта*.

Отличительной особенностью алгоритмов, реализующих метод поиска с возвратом, является наличие точек множественного ветвления, в которых вместо выбора единственного из вариантов продолжения вычислений исследуются все возможные варианты. Если тот или иной вариант не приводит к желаемому результату, то происходит возврат в ближайшую точку ветвления и выбирается один из оставшихся вариантов. Тем самым нужное решение строится не по фиксированным правилам, а *методом проб и ошибок*, т. е. перебором всех возможных вариантов с отбрасыванием непригодных. Отбрасывание непригодных можно производить, используя эвристические соображения, и тем самым сводить количество вычислений к разумным пределам.

Алгоритм решения задачи методом бэктрекинга разделяется на отдельные этапы, которые наиболее естественно описываются с помощью рекурсии. Каждый из этапов завершается одной из трех возможных ситуаций:

- решение найдено;
- достигнута новая точка множественного ветвления с неиспробованными вариантами продолжения;

– достигнута новая точка множественного ветвления, но все варианты продолжения испробованы (*тупиковая ситуация*).

В первом случае алгоритм можно считать завершенным, если нужно было найти единственное решение. Во втором случае алгоритм должен выбрать один из возможных путей для дальнейшего поиска решений. Третий случай – это необходимость вернуться назад к ближайшей точке ветвления (и соответствующему этапу), для которой остались неисследованные пути, и выбрать новую альтернативу. Если такая точка ветвления не будет обнаружена, то решение завершается.

Схема алгоритма с возвратом представлена в листинге 3.10.

Листинг 3.10. Схема алгоритма с возвратом

```
void Attemt(int i, bool hit)
{
   int k = 0;
   do
   {
      //выбор k-ого возможного хода
      if(hit)
      {
         //обработка хода;
         if(i < n) Attemt(i+1, hit);
         if( !hit ) вернуться назад;
      }
   } while(!hit && (k < m);
}
```

Эта схема предполагает, что на каждом шаге существует некоторое фиксированное число путей *m*, а глубина рекурсии ограничена значением *n*. В конкретных программах эта схема может воплощаться различными способами.

3.4.1. Расстановка ферзей

Это классическая задача, на которой обычно демонстрируют применение метода проб и ошибок и алгоритмов бэктрекинга. Она известна как задача о восьми ферзях и формулируется следующим образом: нужно расставить

восемь ферзей на шахматной доске размером 8×8 таким образом, чтобы ни один ферзь не угрожал другому, т. е. чтобы ни на одной из горизонталей, вертикалей или диагоналей не находилось два или более ферзей. Задача в такой формулировке была впервые предложена в 1848 г. немецким шахматистом М. Беццелем. Ее решением занимались многие математики, в том числе К.Ф. Гаусс, который в 1850 году опубликовал 72 варианта такой расстановки.

Исследуем две постановки этой задачи:

- найти некоторое решение;
- найти все возможные решения, то есть все правильные варианты расстановки ферзей.

Для решения первого варианта задачи воспользуемся рекурсивной схемой, приведенной в листинге 3.10:

```
void Attemt(int i, out bool found)
{
   // инициализировать выбор позиции для i-ого ферзя;
   do
   {
      found = false;
      // выбор позиции i-ого ферзя
      if(выбрана позиция k)
      {
         // поставить ферзя;
         if(i < 8)
            Attemt(i+1, out found);
            if(!found)
               // снять ферзя;
            else
               found = true;
      }
   } while(!found && (k < 8);
}
```

Прежде всего, необходимо выбрать способ представления шахматной доски в памяти компьютера. Использование в таком качестве двумерного массива связано с необходимостью при выборе позиции для очередного ферзя проверять, свободны ли соответствующие диагонали и горизонталь. Это в значительной степени усложнит решение. Поскольку позиции ферзей задаются номерами соответствующих горизонталей и вертикалей, то их можно хранить в одномерном массиве, используя в качестве индекса номер вертикали, а в качестве значения – номер горизонтали. Для проверки возможности размещения ферзя в некоторой позиции выбранной вертикали необходимо иметь информацию о занятости горизонтали и двух диагоналей, проходящих через эту позицию. С этой целью объявим три дополнительных булевских одномерных массива, в элементы которых будут записываться значения *true* или *false* в зависимости от состояния занятости соответствующей горизонтали или диагонали. Будем использовать для обозначения размера шахматной доски константу *Size*. Тогда число диагоналей каждого вида (параллельных главной и побочной диагонали соответственно) будет равно *2×Size+1*. Каждая из диагоналей, параллельных главной диагонали, может характеризоваться значением разности координат *i–j*, постоянной для всех ее элементов. Аналогично, в пределах каждой из диагоналей, параллельных побочной, постоянна сумма *i+j* координат ее элементов. С учетом перечисленных обстоятельств массивы, используемые для представления шахматной доски, опишем следующим образом:

```
const int Size = 8;       // размер шахматной доски

     // позиция ферзя на вертикали

RangeArray PosQ = new RangeArray(1, Size);

     // отсутствие ферзя на горизонтали

RangeArrayBool NoHor = new RangeArrayBool(1, Size);

     // отсутствие ферзя на антидиагонали

RangeArrayBool NoRDiag = new RangeArrayBool(2,
2*Size);

     // отсутствие ферзя на диагонали

RangeArrayBool NoLDiag = new RangeArrayBool(-Size+1,
Size-1);
```

Необходимо предусмотреть выход из рекурсии после успешного размещения последнего из ферзей. Поэтому в число параметров рекурсивной функции расстановки ферзей необходимо включить параметр, фиксирующий факт обнаружения решения. Данная функция представлена в листинге 3.11.

Листинг 3.11. Восемь ферзей. Поиск одного варианта

```
void Queen(int i, out bool found)
{
   int j = 0;
   do
   {
      found = false;
      j++;
      if (NoHor[j] && NoRDiag[i+j] && NoLDiag[i-j])
      {
         PosQ[i] = j;  // поставить ферзя
         NoHor[j] = false;
         NoLDiag[i - j] = false;
         NoRDiag[i + j] = false;
         if (i < Size)
         {
            Queen(i + 1, out found);
            if (!found)
            {
               PosQ[i] = 0;         // убрать ферзя
               NoHor[j] = true;  // линии свободны
               NoLDiag[i - j] = true;
               NoRDiag[i + j] = true;
            }
         }
         else
            found = true;
      }
```

```
        }
        while (!(found || (j == Size)));
    }
```

В проекте, форма которого представлена на рисунке 3.5, реализован алгоритм бэктрекинга для данной задачи. На форме позиции ферзей отмечены символом *'W'*.

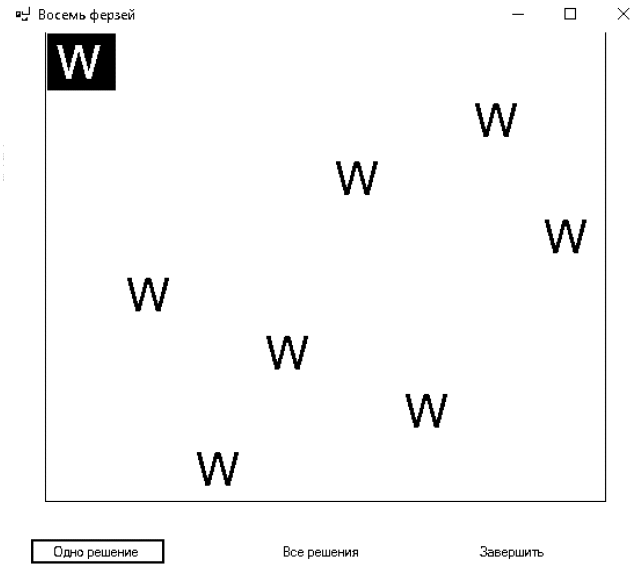


Рис. 3.5. Форма проекта «Восемь ферзей». Реультат поиска единственного решения

На форме размещены три элемента управления:

- элемент *dataGridView1*;
- кнопка «Одно решение»;
- кнопка «Все решения».

В листинге 3.12 представлен обработчик события клик по кнопке «Одно решение». После присваивания всем элементам булевских массивов значения *false* в нем производится вызов рекурсивной функции расстановки ферзей *Queen*.

Листинг 3.12. Восемь ферзей. Расстановка ферзей

```
private void btnSet_Click(object sender,
    EventArgs e)
{
   for (int i=1; i<=Size; i++)
      NoHor[i] = true;    // горизонтали свободны
   for (int i = 2; i <= 2*Size; i++)
      NoRDiag[i]=true; //все антидиагонали свободны
   for (int i =-Size+1; i <= Size-1; i++)
      NoLDiag[i]=true;  // все диагонали свободны
   Queen(1, out ok);
   for (int k = 0; k < Size; k++)
      dgw[k, PosQ[k + 1] - 1].Value = 'W';
}
```

Решение второго варианта задачи о расстановке ферзей – поиска всех возможных конфигураций гораздо проще (листинг 3.13). Оно сводится к реализации общей схемы алгоритма поиска с возвратом, представленной в листинге 3.13.

Листинг 3.13. Восемь ферзей. Поиск всех вариантов

```
void Attemt(int i, ref int n)
{
 // инициализировать выбор позиции для i-ого ферзя;
   for(int j = 0; j < Size; j++)
   {
      // выбор позиции i-ого ферзя
      if(NoHor[j] && NoRDiag[i+j] && NoLDiag[i-j])
      {
         PosQ[i] = j;     / поставить ферзя
```

```
            NoHor[j] = false;
            NoLDiag[i - j] = false;
            NoRDiag[i + j] = false;
            if(i < Size)
               Attemt(i+1, ref n);
            else
               ++n;
            NoHor[j] = true;    // линии опять свободны
            NoLDiag[i - j] = true;
            NoRDiag[i + j] = true;
         }
      }
   }
```

Параметрами рекурсивной функции являются номер *i* ферзя, для которого надо найти позицию, и переменная *n,* используемая для подсчета общего числа решений. Цикл обеспечивает перебор всех возможных позиций для каждого из ферзей и, тем самым, нахождение всех возможных расстановок.

3.4.2. Задача оптимального выбора

Еще одним примером применения метода поиска с возвратом является *задача оптимального выбора*, т. е. поиска решения, которое может считаться *оптимальным*, удовлетворяющим некоторому, наперед заданному условию.

Наиболее очевидной представляется стратегия, основанная на последовательном переборе возможных решений с отбором на каждом этапе наилучшего из уже просмотренных. Здесь очевидна аналогия с поиском минимального элемента в массиве, при котором очередной элемент сравнивается с минимальным из ранее просмотренных и при необходимости этот «текущий» минимум обновляется.

Примером задачи оптимального выбора является хорошо известная задача об упаковке рюкзака, формулируемая следующим образом. Имеется некоторое количество предметов, каждый из которых имеет определенные вес и стоимость. Задача состоит в том, чтобы отобрать предметы, имеющие наибольшую суммарную стоимость при заданном ограничении на их суммарный вес. Такой набор предметов будем называть *оптимальной выборкой.*

Выборки, составляющие приемлемые решения, строятся постепенно, по мере рассмотрения отдельных предметов. Для каждого из предметов можно принять решение о его *включении* либо *невключении* в выборку. Решение о *включении* очередного предмета принимается, если после добавления этого предмета суммарный вес выборки не превысит предельного веса, но ее суммарная стоимость станет больше ранее достигнутого максимума. Если добавление любого из оставшихся предметов ведет к превышению предельного веса, то формирование выборки завершается.

Данные, необходимые для решения этой задачи, описываются следующим образом:

```
struct TItem  // предмет
{
    public int weight;          // вес
    public int cost;            // цена
}
static int n = 15;              // количество предме-
тов
TItem[] a = new TItem[n];       // набор предметов
int maxWeight;    // ограничение на вес выборки
int maxCost; // текущая наибольшая стоимость выборки
int totCost;      // текущая стоимость выборки
int s = 0;        // текущая выборка
int sOpt = 0;     // оптимальная выборка
```

В переменных целого типа *s* и `sOpt` *i*-й бит отвечает за включение или не включение *i*-го предмета. Для хранения в памяти переменной целого типа требуется 4 байта или 32 бита. Предмет с номером *i* входит в множество, если *i*-й бит равен 1. Таким образом, целое позволяет работать с множествами до 32 элементов. Для работы с множествами необходимо уметь выполнять три операции:

- *добавление* *i*-го элемента в множество: *s = s | (1 << i);*
- *удаление* *i*-го элемента из множества: *s = s & ~(1<<i);*
- *проверка* вхождения *i*-го элемента в множество: *s & (1<<i))!=0.*

В листинге 3.14 представлена рекурсивная функция `Opt()`, выполняющая поиск оптимальной выборки путем перебора различных вариантов ее формирования.

Листинг 3.14. Оптимальный выбор

```
void Opt(int i, int weight, int cost)
{
   if (weight + a[i].weight <= maxWeight)
   {
     s |= (1 << i); // s=s+[i]; добавление в выборку
     if (i < n-1)
        Opt(i+1, weight+a[i].weight, cost);
     else
        if (cost > maxCost)
        {
           maxCost = cost;//запомнить локальный
//максимум
           sOpt = s;   // запомнить выборку
         }
         s &= ~(1<<i);  //удаление из выборки
    }
    int tmp = cost - a[i].cost;

    if (tmp > maxCost) // попытка добавить предмет i
       if (i < n - 1)
          Opt(i + 1, weight, tmp);
       else
       {
          maxCost = tmp;
          sOpt = s;  // запомнить выборку
       }
}
```

На рисунке 3.6 представлена форма проекта, реализующего алгоритм нахождения оптимальной выборки.

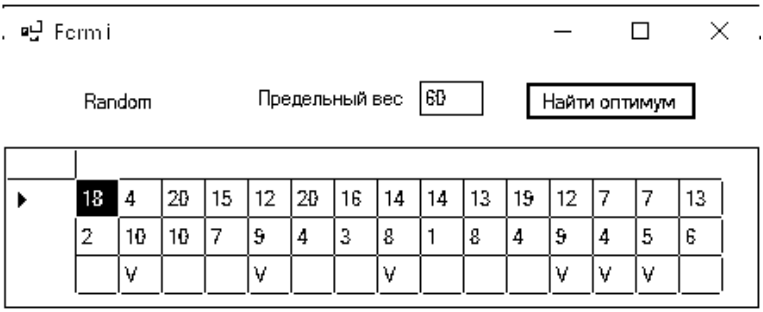


Рис. 3.6. Форма проекта «Оптимальная выборка»

На этой форме размещены следующие элементы управления:

- элемент управления *dataGridView1*, в двух верхних строках которого отображаются вес и цена предметов, а в нижней – предметы, включенные в оптимальную выборку;
- кнопка «Random» для заполнения случайными значениями массивов, хранящих значения весов и стоимостей предметов;
- текстовое поле для ввода предельного веса;
- кнопка «Найти оптимум» для вызова функции *Opt()*, обработчик события клика для которой представлен в листинге 3.15.

Листинг 3.15. Обработчик события клика для кнопки «Найти оптимум»

```
private void buttonOptim_Click(object sender, Even-
tArgs e)
{
   totC = 0;
   for (int i = 0; i < n; i++)
   {       //обновление массива и вычисление полной
стоимости
      a[i].weight = (int)dataGridView1[i,0].Value;
      a[i].cost = (int)dataGridView1[i,1].Value;
      totC = totC+a[i].cost;
```

```
            dataGridView1[i,2].Value = "";
        }
        maxWeight=Convert.ToInt32(textBox1.Text);
    //предельный вес
        maxCost=0;
        s = 0; sOpt = 0;     // инициализация
        Opt(0,0,TotC);
        for (int i=0; i<=n-1; i++)
            if ((sOpt & (1<<i))!=0)
                dataGridView1[i, 2].Value = "V"; ;
    }
```

Здесь предусмотрена возможность задавать значения цены и веса предметов любым из трех способов:

- ручным вводом;
- случайными значениями;
- случайным заполнением с последующим редактированием.

Использование рекурсии существенно ускоряет решение, поскольку прямой перебор потребовал бы вычисления веса и стоимости всех $2^n - 1$ возможных выборок из n предметов. В случае $n = 15$ число выборок оказывается равным 32767. В то же время, благодаря ограничениям, содержащимся в функции *Opt*, число ее рекурсивных вызовов при указанном числе предметов не превышает 500.

Выводы

Рекурсия – это эффективный способ создания изящных и эффективных программ, основанный на альтернативном по отношению к циклам способе повторного выполнения кода. Рекурсивные программы и структуры данных весьма привлекательны, поскольку часто они предоставляют индуктивные аргументы, которые помогают убедиться в правильности и эффективности разработанных программ. Следует, однако, иметь в виду, что применение рекурсии сопряжено с опасностью возникновения ошибок, ведущих к экспоненциальному увеличению количества вызовов функций или недопустимо большой степени вложенности. Поэтому во всех тех случаях, когда существует очевидное итеративное решение задачи, следует избегать применения рекурсии.

Существует обширный класс задач, для которых применение рекурсии принципиально необходимо. К их числу относятся задачи искусственного интеллекта, связанные с поиском решений, оптимальных по тем или иным критериям. Примеры подобных задач были представлены в данной главе.

Упражнения

1. Напишите рекурсивный алгоритм получения всех возможных перестановок N целых чисел.
2. Напишите рекурсивную функцию, проверяющую правильность расстановки скобок в строке. При правильной расстановке выполняются условия:
   - количество открывающих и закрывающих скобок равно;
   - внутри любой пары «открывающая – соответствующая закрывающая скобка» скобки расставлены правильно.
3. Создайте функцию, печатающую все возможные представления натурального числа N в виде суммы других натуральных чисел.
4. Дано натуральное число $N>1$. Предложите рекурсивный алгоритм проверки, является ли оно простым. Алгоритм должен иметь сложность $O(1ogN)$.
5. Дано слово, состоящее только из строчных латинских букв. Проверьте, является ли это слово палиндромом.

Глава 4. Деревья

Предыдущая глава была посвящена рассмотрению рекурсивных алгоритмов. Однако рекурсивный подход может быть применен и для определения структур данных. Наиболее часто используемым видом таких структур являются *деревья*. Деревья имеют широкое применение при реализации трансляторов таблиц решений, при работе с арифметическими выражениями, при создании и ведении таблиц символов, где их используют для отображения структуры предложений, в системах связи для экономичного кодирования сообщений и во многих других случаях.

Деревья наилучшим образом приспособлены для решения задач искусственного интеллекта и *синтаксического анализа*.

4.1. Понятия и определения

Древовидная структура (дерево) определяется следующим образом: *дерево* (*tree*) с базовым типом *T* — это

- либо пустая структура;
- либо узел типа *T*, с которым связано конечное число древовидных структур, называемых *поддеревьями (subtree*).

Если с узлом связаны только два поддерева, то дерево называется *двоичным* (*бинарным)*. В этой главе будут рассматриваться только двоичные деревья.

Для описания деревьев используются термины, заимствованные из ботаники и генеалогии.

«Ботаническими» являются термины:

- *корень* (*root*) – самый «верхний» узел дерева;
- *ветвь* (*brunch*) – цепочка связанных между собой узлов;
- *лист* (*leaf*) – узел, не имеющий поддеревьев.

Термины, заимствованные из генеалогии, описывают отношения:

- *родительским* (*parent*) называется узел, который находится непосредственно над другим узлом;
- *дочерним* (*child*) называется узел, который находится непосредственно под другим узлом;
- *предки* данного узла — все узлы на пути от корня до данного узла;
- *потомки* — все узлы, расположенные ниже данного.

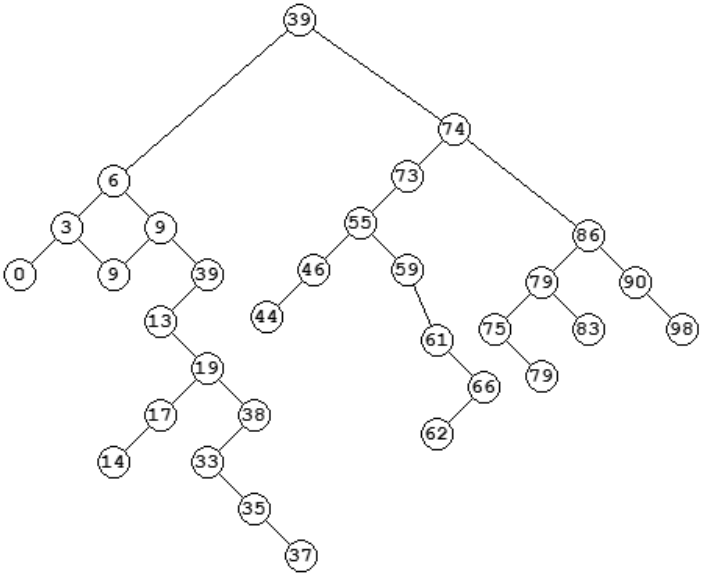


Рис. 4.1. Представление дерева

Специальные термины, связанные с программной реализацией деревьев и рядом специальных задач:

- *узел* (*node*) — это точка, где может возникнуть ветвь;
- *внутренний* узел (*internal node*) — узел, не являющийся ни листом, ни корнем;
- *порядок узла* (*node degree*) — количество его дочерних узлов;
- *глубина* (*depth*) *узла* — количество его предков плюс единица;
- *глубина* (*высота*) *дерева* — максимальная глубина всех узлов;
- *длина пути* к узлу — количество ветвей, которые нужно пройти, чтобы продвинуться от корня к данному узлу;
- *длина пути дерева* — сумма длин путей всех его узлов, называемая также длиной *внутреннего пути*.
- *сестринские* (*братские*) — узлы, у которых один и тот же родитель.

Множество узлов, имеющих одинаковую глубину, образуют *уровень дерева.* Рисунок 4.1 демонстрирует общепринятое изображение дерева, как бы растущего вниз.

4.2. Основные операции с бинарными деревьями

Дерево является динамической структурой данных. В процессе выполнения программы деревья создаются и модифицируются путем добавления и удаления узлов, а также изменения их информационной части. Поэтому узел дерева определяется как структура, содержащая информационную часть и два поля-ссылки, связанные с левым и правым поддеревьями данного узла. При отсутствии соответствующих поддеревьев эти поля получают значение *null*.

В листинге 4.1 приведено описание класса узла дерева с включением дополнительных полей, которые необходимы для графического представления дерева на форме.

Листинг 4.1. Класс узла

```
public class Node// узел
{
    public object data;// значение ключа
    public Node left;  // ссылка на левое поддерево
    public Node right; // ссылка на правое поддерево
    public int x;
    public int y;
    public bool visit; // признак выделенности узла
    public int count; // счетчик повторений значений
    public Node(Node left, Node right, object data,
//конструктор
        int x, int y)
}
```

На основе этого класса можно создать класс дерева (листинг 4.2). Все методы этого класса – рекурсивны.

Листинг 4.2. Класс дерева

```
class MyTree
{
    Node top;                  // вершина дерева
```

```
    public Node SelectNode;    // выделенный узел
    public Bitmap bitmap;      // канва для рисования
    public class Node          // класс узла
    const int step = 30;
    const int Geom = 1;
    Graphics g;
    Pen MyPen;
    SolidBrush MyBrush;
    Font MyFont;
    public void Search(int val)  // поиск и вставка
    public Node FindNode(int x, int y)  // поиск по
координатам
    Node FindNodeVal(int val)  // поиск по значению
    public void Insert(object data, int x,int y)  //
вставка
    void DeSelect() //снятие признака посещения
    public void Delta(Node p, int dx, int dy)  //
смещение поддерева
    public void Delete(Node p)   // удаление узла
    public void Draw()           // изображение де-
рева
}
```

Динамический способ представления дерева – не единственно возможный: дерево можно также представить с помощью массива. Элементы массива будут содержать значения узлов дерева, а с помощью индексов можно установить структуру дерева. Так, в первом элементе i=1 хранится значение корня. Элемент с i=2 хранит значение левого сына корня, элемент с i=3 хранит значение правого сына корня. И так для любого элемента с индексом i индекс его левого сына равен $2 \times i$, а правого — $2 \times i+1$. Индекс отца будет равен $i/2$. Однако так можно хранить деревья со строго определенной структурой. Ниже будет рассмотрен пример сортировки частично упорядоченным деревом, представленным массивом. Для деревьев произвольной формы в массив приходится заносить специальные значения, соответствующие `null`, если у какого-то узла отсутствует тот или иной сын, или для каждого узла хранить

индексы его сыновей. Так, в таблице 4.1 приведен массив, хранящий дерево, представленное на рисунке 4.1. Верхняя строка содержит индексы массива, вторая строка — значения ключей, а третья и четвертая строки — индексы левого и правого сыновей. Значение 0 соответствует отсутствию сына.

Таблица 4.1

Дерево, представленное с помощью массива

Индекс	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Ключ	8	7	9	3	11	2	5	10	15	1	4	6	13	19	12	14
Левый	2	4	0	6	8	10	11	0	13	0	0	0	15	0	0	0
Правый	3	0	5	7	9	0	12	0	14	0	0	0	16	0	0	0

В связи с двоичными деревьями (и деревьями вообще) вводится важная категория алгоритмов: алгоритмы *обхода дерева*. Такой алгоритм – это метод, который позволяет получить доступ к каждому узлу дерева один и только один раз.

Для каждого узла выполняются некоторые виды обработки (проверка, суммирование и т.п.), однако способ обхода не зависит от конкретных действий и является общим для всех алгоритмов обработки узлов.

Отдельные узлы посещаются в некотором определенном порядке. Существуют три порядка, использующие *обход в глубину*:

1. Сверху-вниз (`PreOrder`):
- обработать корень;
- *обход* левого поддерева;
- *обход* правого поддерева;

2. Слева-направо (*InOrder*):
- *обход* левого поддерева;
- обработать корень;
- *обход* правого поддерева;

3. Снизу-вверх (*PostOrder*):
- *обход* левого поддерева;
- *обход* правого поддерева;
- обработать корень.

Четвертый порядок обхода узлов дерева можно назвать *обходом в ширину*. В этом случае сначала обращаются ко всем узлам на данном уровне дерева, а затем переходят к следующему уровню. Обход в ширину часто используется в алгоритмах, осуществляющий полный поиск в дереве. Наиболее

прост алгоритм обхода в ширину при представлении дерева в виде массива – это цикл по всем элементам.

Для дерева, представленного на рисунке 4.1, четыре способа обхода дают следующий результат:

1) 8 7 3 2 1 5 4 6 9 11 10 15 13 12 14 19
2) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 19
3) 1 2 4 6 5 3 7 10 12 14 13 19 15 11 9 8
4) 8 7 9 3 11 2 5 10 15 1 4 6 13 19 12 14

Первые три метода легко представить в виде рекурсивных процедур. Процедуры просты, и это свидетельство того, что рекурсивные структуры естественнее всего описывать рекурсивными алгоритмами.

Процедуры представлены в листинге 4.3. Функция *Proc* выполняет некоторые действия для конкретного узла. Учитывается приведенное выше описание узла дерева.

Листинг 4.3. Процедуры обхода дерева

```
void PreOrder(TNode Tree)  // Сверху-вниз
{
  if (Tree != null)
  {
    Proc(Tree); // обработка узла
    PreOrder(Tree.left);
    PreOrder(Tree.right);
  }
}

void InOrder(TNode Tree);  //  Слева-направо
{
  if (Tree != null)
  {
    InOrder(Tree.left);
    Proc(Tree); // обработка узла
    InOrder(Tree.right);
  }
}
```

```
void PostOrder(TNode Tree);  // Снизу-вверх
{
  if (Tree != null)
  {
    PostOrder(Tree^.left);
    PostOrder(Tree^.right);
    Proc(Tree^); // обработка узла
  }
}
```

Примечание

Следует обратить внимание, что ссылка *Tree* передается как параметр-значение. Это отражает тот факт, что здесь существенна сама *ссылка* (указатель) на рассматриваемое поддерево, а не переменная, значение которой есть эта ссылка и которая могла бы изменить значение, если бы *Tree* передавалась как параметр-переменная.

Пример процедуры, осуществляющий обход дерева, приведен ниже, в листинге 4.4. Обход дерева осуществляется с целью найти ближайший к указателю мышки узел нарисованного на канве дерева.

4.2.1. Упорядоченные деревья

Упорядоченным называется дерево, у которого для каждого узла *Node* значение левого дочернего узла меньше, чем значение в *Node*, а значение правого дочернего узла больше значения в *Node*. Если в дереве могут содержаться одинаковые значения, то программист сам должен определить, влево или вправо помещать значение, равное значению в родительском узле, то есть соответствующее строгое неравенство заменить на нестрогое.

Построение упорядоченного дерева начинается с корня. Первое из входящих значений и будет значением корня. Для каждого следующего значения производится, начиная с корня, сравнение со значением в очередном узле. Если это значение не превосходит значения в узле, то переходим в левое поддерево, иначе переходим в правое. Эти переходы и сравнения продолжаются до тех пор, пока не придем к ссылке, которая равна *null*. Тогда создается новый узел. Этот алгоритм, реализованный в виде рекурсивной процедуры, приведен в листинге 4.4.

Листинг 4.4. Функция добавления узла в упорядоченное дерево

```
public void Insert(ref Node t, int data, int x, int
y)
```

```
// вставка
{
  if (t == null)
    t = new Node(null, null, data, x, y);
  else
    if (data <= Convert.ToInt32(t.data))
      Insert(ref t.left, data, t.x - step,
          t.y + dh * step);
    else
      Insert(ref t.right, data, t.x + step,
          t.y + dh* step);
}
```

Следует отметить, что в данной функции параметр *t* передается как параметр-переменная, а не как параметр-значение в функциях листинга 4.1. Это существенно, поскольку в случае включения нового узла параметру-переменной должно присваиваться новое значение – ссылка на включенный узел, старое значение параметра было равно *null*.

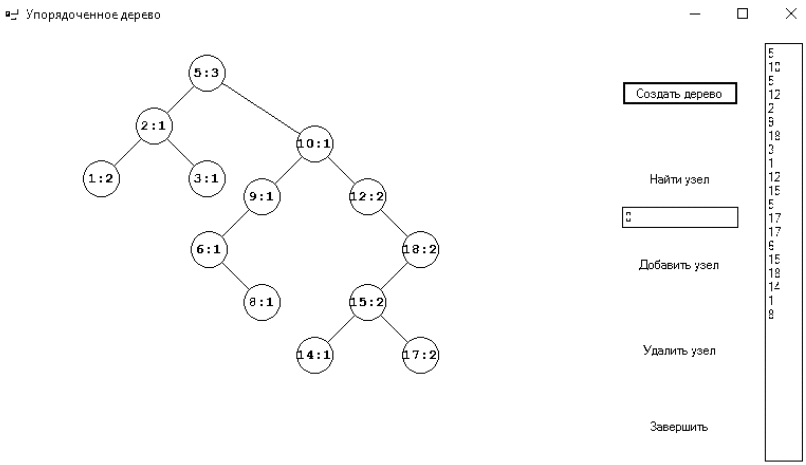


Рис. 4.2. Форма для работы с деревьями

На рисунке 4.2 приведена форма проекта «Упорядоченное дерево», содержащая следующие элементы управления:

– текстовое поле, в котором вводятся значения для узлов дерева;

- кнопка «Создать дерево», при нажатии на которую генерируется последовательность случайных чисел, на ее основе создается упорядоченное дерево и строится его изображение;
- кнопка «Найти узел», при нажатии на которую ищется узел дерева с указанным значением;
- кнопка «Добавить узел», при нажатии на которую в дерево добавляется узел с указанным значением информационного поля;
- кнопка «Удалить узел», при нажатии на которую удаляется узел с указанным значением;

Построение упорядоченного дерева производится с помощью приведенной выше процедуры *Insert()*.

Листинг 4.5. Создание дерева

```
private void btnCreate_Click(object sender, Even-
tArgs e)
{
   int L = textBox1.Lines.Count();
   for (int i = 0; i < L; i++)
   {
      if (textBox1.Lines[i] != "")
      {
         int k = Convert.ToInt32(textBox1.Lines[i]);
         myTree.Insert(ref myTree.top, k, 200, 40);
      }
   }
   MyDraw();
}
```

Рисование же дерева использует алгоритм обхода дерева способом «сверху-вниз», поэтому построение изображения дерева начинается с его корня. Затем рисуются узлы левого поддерева, затем правого. При прорисовке узла запоминаются его координаты, которые используются как при прорисовке ветвей текущего дерева, так и при повторной прорисовке дерева, если менять мышкой положение его узлов на форме. Основные процедуры приведены в листинге 4.6.

Листинг 4.6. Рисование дерева

```
void DrawNode(Node p)          // изображение дерева
{
   int R = 12;
   if (p.left != null)
      g.DrawLine(MyPen,p.x, p.y,p.left.x,p.left.y);
   if (p.right != null)
      g.DrawLine(MyPen, p.x, p.y, p.right.x,
          p.right.y);
   if (p.visit)
      MyBrush = (SolidBrush)Brushes.Yellow;
   else
      MyBrush = (SolidBrush)Brushes.LightYellow;
   g.FillEllipse(MyBrush, p.x - R, p.y - R,
      2*R, 2*R);
   g.DrawEllipse(MyPen, p.x - R, p.y - R, 2*R, 2*R);
   string s = Convert.ToString(p.data);
   SizeF size = g.MeasureString(s, MyFont);
   g.DrawString(s, MyFont, Brushes.Black,
      p.x - size.Width / 2, p.y - size.Height / 2);
   if (p.left != null)
      DrawNode(p.left);
   if (p.right != null)
      DrawNode(p.right);
}
```

В процессе рисования один узел может наложиться на другой. Поэтому в программе предусмотрено перемещение узла с помощью мыши. Соответствующие обработчики событий приведены в листинге 4.7.

Листинг 4.7. Перемещение мышью узла дерева

```
private void FormMain_MouseDown(object sender,
    MouseEventArgs e)
{
   myTree.DeSelect(myTree.top);
   myTree.SelectNode =
      myTree.FindNode(myTree.top, e.X,e.Y);
   drawing = myTree.SelectNode != null;
   if (drawing)
      myTree.SelectNode.visit = true;
}

private void FormMain_MouseMove(object sender,
    MouseEventArgs e)
{
   if (drawing)
     myTree.Delta(myTree.SelectNode,
        myTree.SelectNode.x –
        e.X, myTree.SelectNode.y - e.Y);
   else
   {
      myTree.DeSelect(myTree.top);
      myTree.SelectNode=myTree.FindNode(myTree.top,
         e.X,e.Y);
      if (myTree.SelectNode != null)
         myTree.SelectNode.visit = true;
   }
   MyDraw();
```

```
}

private void FormMain_MouseUp(object sender,
    MouseEventArgs e)
{
    drawing = false;
}
```

Обработчик события *FormMain_MouseDown*, выполняющийся при нажатии на левую кнопку мыши, вызывает метод *FindNode()*, в котором производится поиск узла, который будет перемещаться. Этому методу в качестве параметров передаются координаты указателя мыши (листинг 4.8).

Листинг 4.8. Поиск узла по координатам

```
public Node FindNode(Node p, int x, int y)
// поиск по координатам
{
   Node result = null;
   if (p == null)
      return result;
   if (((p.x -x) * (p.x - x) +
      (p.y - y) * (p.y - y)) < 100)
      result = p;
   else
   {
      result = FindNode(p.left, x, y);
      if (result == null)
         result = FindNode(p.right, x, y);
   }
   return result;
 }
```

Метод *FindNode()* представляет собой рекурсивную процедуру обхода дерева с поиском узла, расстояние от которого до координат указателя мыши минимально. В завершение работы метод *FormMain_MouseDown* устанавливает флажок разрешения перемещения узла *drawing* = *true*.

Обработчик события *FormMain_MouseMove,* связанный с перемещением мыши при нажатой левой кнопке, обращается к методу *Delta()*, изменяющему координаты выбранного узла и всех его потомков. Затем заново строится изображение дерева. Метод *Delta()* представлен в листинге 4.9.

Листинг 4.9. Изменение координат узла и его потомков

```
public void Delta(Node p, int dx, int dy)
// смещение поддерева
{
    p.x -= dx; p.y -= dy;
    if (p.left != null)
       Delta(p.left, dx, dy);
    if (p.right != null)
       Delta(p.right, dx, dy);
}
```

4.2.2. Поиск по дереву с включением

Эффективность использования структуры упорядоченное дерево можно наглядно продемонстрировать на примере задачи о построении частотного словаря. Суть этой задачи заключается в построении на основе заданного текста упорядоченной последовательности входящих в него слов с указанием для каждого из них кратности вхождения.

С помощью упорядоченного дерева эту задачу можно решить следующим образом. Последовательно перебирая слова текста, строим дерево, каждый узел которого содержит два информационных поля: слово и кратность его вхождения. Для каждого очередного слова производится поиск в дереве. Если слово найдено, то увеличивается значение поля кратности вхождения у соответствующего узла. В противном случае для этого слова создается новый узел со значением поля кратности, равным 1. Этот процесс называется *поиском по дереву с включением*.

Алгоритм поиска по дереву с включениями приведен в листинге 4.10.

Листинг 4.10. Поиск по дереву с включением

```
public void Insert(ref Node t, int data,
    int x, int y)
// вставка
{
   if (t == null)
      t = new Node(null, null, data, x, y);
   else
      if (data < Convert.ToInt32(t.data))
         Insert(ref t.left, data, t.x-step,
                t.y+ dh * step);
      else
         if (data > Convert.ToInt32(t.data))
            Insert(ref t.right, data, t.x + step,
                t.y + dh * step);
         else
            t.count++;
}
```

Как следует из листинга 4.10, поиск осуществляется перемещением по узлам дерева с выбором на каждом этапе дальнейшего направления движения в зависимости от соотношения между искомым и хранящимся в текущем узле значением. После завершения процесса построения дерева можно выполнить его обход по способу «сверху-вниз» с формированием упорядоченной последовательности значений.

В листинге 4.11 представлен алгоритм обхода упорядоченного дерева с формированием последовательности значений, упорядоченной по убыванию.

Листинг 4.11. Формирование отсортированной последовательности

```
public string SetStrSort(Node p)
{
   string s="";
```

```
        if (p == null)
            return s;
        if (p.right != null)
            s += SetStrSort(p.right);
        s += Convert.ToString(p.data) + "\r\n";
        if (p.left != null)
            s += SetStrSort(p.left);
        return s;
    }
```

Результат выполнения этого алгоритма представлен на рисунке 4.3.

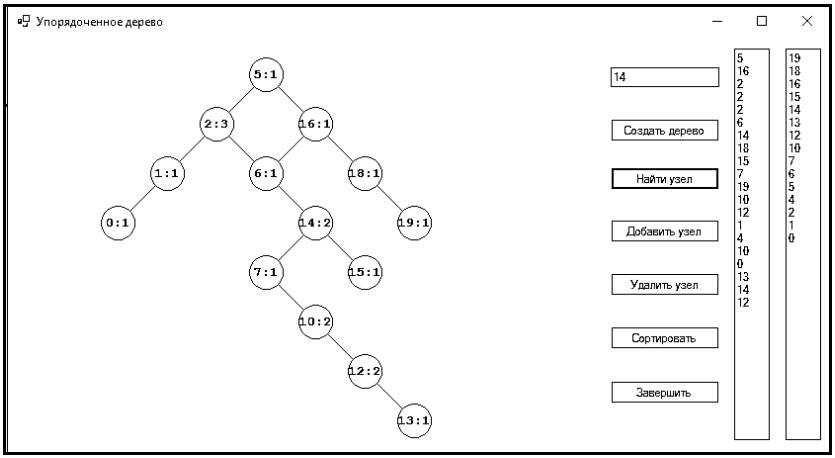


Рис. 4.3. Вывод отсортированной последовательности

4.2.3. Удаление из упорядоченного дерева

Задача удаления узла из упорядоченного дерева является обратной задаче добавления узла. После операции удаления дерево должно остаться упорядоченным. Операция удаления узла из упорядоченного дерева в общем случае является более сложной, чем добавление узла. Различные ситуации, возникающие при удалении узла, иллюстрируются рисунком 4.4.

Удаление узла легко выполняется, если этот узел является листом дерева (узел 13 на рис. 4.4). Лист дерева удаляется без перестройки дерева. Если удаляемый узел имеет только одного сына, то удаляемый узел заменяется узлом-сыном (узел 15 на рис. 4.4). Трудность заключается в удалении узла с

двумя сыновьями, поскольку невозможно указать одной ссылкой два направления (узел 5 на рис. 4.4). В этом случае удаляемый узел нужно заменить либо на *самый правый элемент его левого поддерева*, либо на *самый левый элемент его правого поддерева*. Причина такого выбора заключается в том, что указанные узлы не могут иметь более одного потомка, что позволяет «прикрепить» к ним «осиротевшее» поддерево.

Так, при замене удаляемого узла на самый правый элемент левого поддерева (узел 10 на рис. 4.4) мы сможем «прикрепить» к нему правое поддерево удаленного узла, а его собственное левое поддерево передать в качестве правого поддерева его узлу-родителю. При альтернативном выборе ситуация будет симметричной.

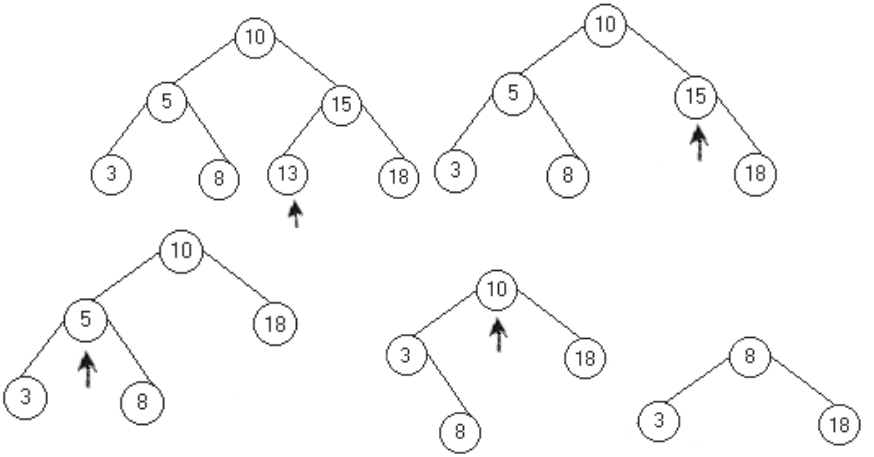


Рис. 4.4. Удаление из упорядоченного дерева

Алгоритм удаления узла с ключом *x* из упорядоченного дерева реализован процедурой `Delete()`, приведенной в листинге 4.12.

Функция различает три случая:

- в дереве нет узла с ключом *x*;
- узел с ключом *x* имеет не более одного сына;
- узел с ключом *x* имеет двух сыновей.

Листинг 4.12. Удаление из упорядоченного дерева

```
public void Delete(int data, ref Node tree)
{
```

```
    if (tree != null)
        if (data < Convert.ToInt32(tree.data))
            Delete(data, ref tree.left);
        else
            if (data > Convert.ToInt32(tree.data))
                Delete(data, ref tree.right);
            else   {
               q = tree;
               if (q.right == null)
                  tree = q.left;
               else
                  if (q.left == null)
                     tree = q.right;
                  else
                     Del(ref q.left);
            }
}
```

Вспомогательная рекурсивная функция *Del* вызывается только в случае, когда удаляемый узел имеет двух сыновей.

Листинг 4.13. Вспомогательная функция удаления

```
void Del(ref Node r)
{
   if (r.right != null)
      Del(ref r.right);
   else
   {
      q.data = r.data; q = r; r = r.left;
   }
}
```

Эта функция обеспечивает движение вниз до конца самой правой ветви левого поддерева удаляемого узла *q* и затем заменяет ключ в *q* соответствующим значением самого правого узла *r* этого левого поддерева. После чего *r* удаляется.

Отметим, что структура упорядоченного дерева зависит от порядка добавления элементов. Высокие и «тонкие» деревья, подобные представленному на рисунке 4.5 слева, могут иметь высоту порядка числа узлов *N*. Добавление элемента в такое дерево или поиск в нем могут потребовать порядка *N* операций, поэтому такие деревья, представляющие собой сложную форму списка, неэффективны для работы. Для эффективного использования древовидной структуры желательно иметь упорядоченные и *сбалансированные* деревья, то есть деревья, в которых у каждого узла левое и правое поддерево содержат примерно одинаковое число узлов. Существуют хорошо известные алгоритмы балансировки [12], но здесь они не рассматриваются.

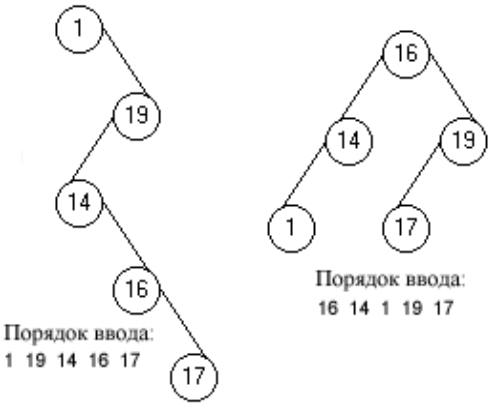


Рис. 4.5. Деревья, построенные при различном порядке поступления элементов

4.3. Турнирная сортировка

Простейший вид сортировки на деревьях уже упоминался – это построение упорядоченного дерева и последовательное взятие значений узлов при обходе «слева-направо». Однако существуют и более эффективные сортировки, использующие древовидную структуру.

Идея турнирной сортировки фактически заимствована у организаторов спортивных чемпионатов. Чемпионат проводится в несколько туров, причем в следующий тур выходят победители парных встреч из предыдущего тура.

Турнирная сортировка является одним из вариантов *пирамидальной сортировки*. Алгоритм такой сортировки состоит из двух шагов. Сначала на основе сортируемой последовательности строится дерево, называемое *пирамидой,* и с этим связано название алгоритма (*HeapSort* от англ. *heap* – куча, пирамида).

На рисунке 4.6 показана форма проекта «Турнирная сортировка» с изображенным на ней пирамидальным деревом. На этой форме размещены следующие элементы управления:

- текстовое поле, предназначенный для ввода данных;
- кнопка «Создать», при нажатии на которую строится исходное турнирное дерево, имеющее вид пирамиды, и отображается на форме;
- кнопка «Сортировать», при нажатии на которую выполняется выбор значения с вершины дерева и перестройка дерева;
- элемент `PictureBox` для построения изображения дерева.

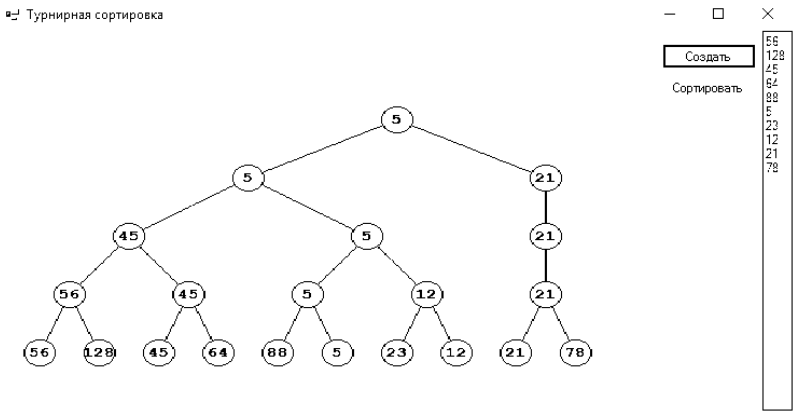


Рис. 4.6. Исходное дерево турнирной сортировки

В отличие от обычного дерева пирамида формируется от основания к вершине. Кроме того, узлы, расположенные на одном уровне, связываются в линейный список. По этой причине каждый узел дерева, дополнительно к ссылкам на прямых потомков, содержит указатель на брата. Структура класса, описывающего узел дерева-пирамиды, представлена в листинге 4.14.

Листинг 4.14. Класс Node

```
public class Node
{
```

```
    public int data;   // данные
    public Node left;
    public Node right; // потомки
    public Node next;  // брат
    public int x;
    public int y;      // координаты узла
    public Node(int data, Node left, Node right,
       int x, int y)
}
```

На первом этапе строится линейный список из элементов сортируемой последовательности. Следующий уровень пирамиды формируется путем сравнения ключей в каждой из пар и выбора элемента с меньшим значением ключа. Процесс продолжается до тех пор, пока не будет достигнут уровень, содержащий единственный элемент с минимальным значением ключа.

Для построения пирамиды необходимо, чтобы все значения были известны. Поэтому вводимые значения считываются из текстового поля и заносятся в массив. При построении пирамиды одновременно производится прорисовка ее на форме. В листинге 4.15 приведено описание метода *HeapCreate()*, в котором производится построение пирамиды и вывод ее графического изображения на экран. Входным параметром этого метода является высота области построения изображения пирамиды *H*, а результатом – узел на вершине пирамиды.

Листинг 4.15. Создание турнирной пирамиды

```
public Node HeapCreate(int H)
// Создание дерева-пирамиды
{
   Node pLevel; // первый узел текущего уровня
   Node pNew, pOld;   // рабочие переменные
   Node comp1, comp2;  // узлы соревнующейся пары
     // построение самого нижнего уровня пирамиды
   int x = deltaX;
   int y = H-diametr-deltaY;
```

```
  // создание первого элемента
pLevel = new Node(a[0],null,null,x,y);
pOld = pLevel;
pNew = null;
  // включение в список элементов массива
for (int i = 1; i < a.Length; i++)
{
  x = x + deltaX + radius;
  pNew = new Node(a[i],null,null,x,y);
  pOld.next= pNew; // линейный список по уровню
  pOld = pNew;
}
pNew.next = null;
  // построение других уровней
while (pLevel.next != null)
{  //цикл до вершины пирамиды
   y = y - deltaY - diametr;
   comp1 = pLevel;
   pLevel = null;   //начало нового уровня
   while (comp1 != null)
   {      // цикл по очередному уровню
      comp2 = comp1.next;
        // адреса потомков из предыдущего уровня
      pNew = new Node(0,comp1,comp2,x,y);
      // связывание в линейный список по уровню
      if (pLevel==null)
         pLevel = pNew; // назначение pLevel
      else
         pOld.next = pNew;
```

```
                pOld = pNew;
                   // состязание данных за выход на уровень
                if ((comp2 == null) ||
                     (comp2.data > comp1.data))
                   pNew.data = comp1.data;
                else
                   pNew.data = comp2.data;
                if (comp2==null)
                   x = comp1.x;
                else
                   x = (comp1.x+comp2.x)/2;
                pNew.x = x; pNew.y = y;
                   // переход к следующей паре
                if (comp2!=null)
                   comp1 = comp2.next;
                else
                   comp1 = null;
             }
          }
          return pLevel;
       }
```

Метод *HeapCreate()* является членом класса *MyTour*, который кроме того содержит открытое целочисленное поле-массив *a* и поле-массив *arr* экземпляров класса *Node*. В массиве *a* хранится исходная последовательность значений, а в массиве *arr* – отсортированная последовательность узлов дерева-пирамиды. Членами класса *MyTour* являются методы, выполняющие построение изображения пирамиды – *Draw(), DrawNode()* и *SetDelta()*.

Обработчик события клика по кнопке «Создать», представленный на листинге 4.16, считывает данные из текстового поля *textBox1* на форме, сохраняя их в поле-массиве *a*, вызывает метод *HeapCreate()* и затем метод *Draw()* (листинг 4.16).

Листинг 4.16. Вызов метода HeapCreate()

```
private void buttonCreate_Click(object sender, Even-
tArgs e)
{
   int L = textBox1.Lines.Count();
   myTour.a = new int[L];
   myTour.arr = new MyTour.Node[0];
   for (int i = 0; i < L; i++)
     if (textBox1.Lines[i] != "")
        myTour.a[i] = Con-
vert.ToInt32(textBox1.Lines[i]);
   myTour.SetDelta(ClientRectangle.Width –
panel1.Width);
   myTour.head = my-
Tour.HeapCreate(ClientRectangle.Height);
   myTour.Draw();
   g.DrawImage(myTour.bitmap, ClientRectangle);
}
```

Метод *DrawNode()*, представленный листингом 4.17, реализует рекурсивный алгоритм построения изображения дерева и выполняет прорисовку отдельных узлов с выводом в них значений ключей. В этом же листинге представлен метод *Draw()*, из которого производится вызов *DrawNode()*.

Листинг 4.17. Метод рисования узла

```
void DrawNode(Node p) // рекурсивное рисование узлов
{
   if (p.left!=null)
   g.DrawLine(MyPen, p.x,p.y,p.left.x,p.left.y);
   if (p.right != null)
      g.DrawLine(MyPen, p.x, p.y, p.right.x,
p.right.y);
```

```
   MyBrush.Color = Color.LightYellow;
   g.FillEllipse(MyBrush, p.x - radius,
      p.y - radius, 2 * radius, 2 * radius);
   g.DrawEllipse(MyPen, p.x - radius, p.y - radius,
                2 * radius, 2 * radius);
   string s = Convert.ToString(p.data);
   SizeF size = g.MeasureString(s, MyFont);
   g.DrawString(s, MyFont, Brushes.Black,
       p.x - size.Width / 2, p.y - size.Height / 2);
   if (p.left != null)
       DrawNode(p.left);
   if (p.right != null)
       DrawNode(p.right);
}

public void Draw()          // рисование дерева
{
   using (g = Graphics.FromImage(bitmap))  {
     Color cl = Color.FromArgb(255, 255, 255);
     g.Clear(cl);
     MyPen = Pens.Black;
     if (head != null)
        DrawNode(head);
     // вывод упорядоченной последовательности узлов
     int L = arr.Length;
     for (int i = 0; i < L; i++)  {
        g.FillEllipse(MyBrush, arr[i].x - radius,
            arr[i].y - radius, 2 * radius, 2 * ra-
dius);
```

```
            g.DrawEllipse(MyPen, arr[i].x - radius,
                arr[i].y - radius, 2 * radius, 2 * ra-
dius);
            string s = Convert.ToString(arr[i].data);
            SizeF size = g.MeasureString(s, MyFont);
            g.DrawString(s, MyFont, Brushes.Black,
               arr[i].x-size.Width/2,
               arr[i].y - size.Height/2);
         }
      }
   }
```

После построения дерева-пирамиды начинается этап формирования упорядоченной последовательности. Он состоит из отдельных шагов, на каждом из которых узел с минимальным значением ключа удаляется из вершины пирамиды и добавляется в отсортированную последовательность узлов *arr*. Это действие требует полной перестройки пирамиды, поскольку значение, удаляемое из пирамиды, представлено соответствующим узлом на каждом ее уровне. Алгоритм перестройки пирамиды осуществляет рекурсивный спуск до нижнего уровня по ветви дерева, состоящей из узлов со значениями, равными минимальному значению ключа. Узел-лист с минимальным значением удаляется присваиванием ссылке на него значения *null*, а во все подобные узлы на вышележащих уровнях записывается минимальное из значений в их дочерних узлах. Перестройка пирамиды производится методом *Competition()* класса *MyTour*, единственным параметром которого является указатель на вершину дерева-пирамиды, передаваемый по ссылке.

Метод, выполняющий перестройку дерева-пирамиды, приведен в листинге 4.18.

Листинг 4.18. Метод перестройки пирамиды

```
public Node Competition(ref Node ph)
// Реорганизация поддерева
{
   if (ph.left!=null)
   {
     if (ph.left.data==ph.data)
```

```
            ph.left = Competition(ref ph.left);
          else
            ph.right = Competition(ref ph.right);
        }
        else
          if (ph.right!=null)
            ph.right = Competition(ref ph.right);

        if ((ph.left==null) && (ph.right==null))
          ph = null;
        else // состязание данных сыновей
          if ((ph.left==null) || ((ph.right != null)
               && (ph.left.data>ph.right.data)))
            ph.data = ph.right.data;
          else
            ph.data =ph.left.data;
          return ph;
    }
```

На рисунке 4.7 демонстрируется результат реорганизации пирамиды после выбора двух узлов.

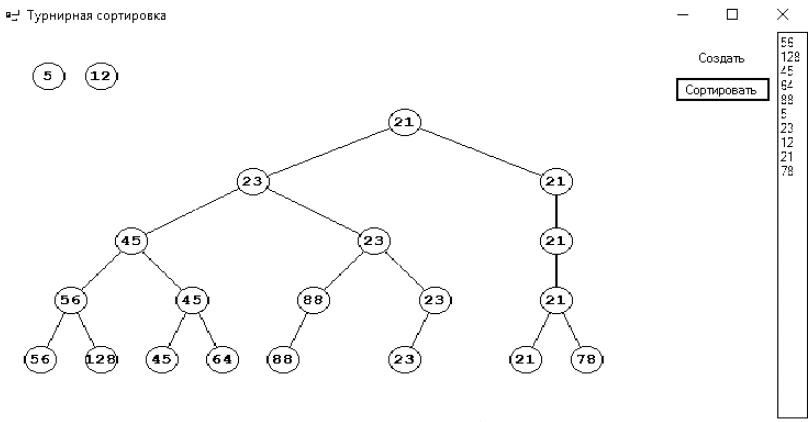


Рис. 4.7. Выборка значений из турнирной пирамиды

Для оценки временной эффективность данного алгоритма сортировки следует принять во внимание, что для построения дерева требуется выполнить $N-1$ сравнений, а для формирования упорядоченной выборки узлов – $N \times log_2(N)$ сравнений. Таким образом, скорость роста алгоритма, определяющаяся наибольшей из этих величин, составит $O(N \times log_2 N)$. Однако сложность выполнения операций над динамическими структурами данных не позволяет получить существенного преимущества во времени выполнения перед алгоритмами сортировки для статических структур данных. Недостатком алгоритма является также избыточное использование, обусловленное кратным присутствием в дереве-пирамиде узлов с одинаковыми значениями ключа. Фактически это приводит к удвоению числа узлов по сравнению с размером сортируемой последовательности.

4.4. Основы работы интерпретатора

Интерпретатор — это программа, которая переводит исходный текст в некоторое промежуточное представление, а потом выполняет соответствующие действия.

Компилятор транслирует программу, написанную на некотором языке высокого уровня, в объектный код.

Процесс проектирования и создания компилятора заслуживает отдельной книги. Однако можно рассмотреть основные положения работы компилятора на примере построения *синтаксического дерева* или *дерева разбора* арифметического выражения. И построение дерева разбора, и обход его для вычисления выражения — еще один пример использования деревьев.

Работа компилятора складывается из двух основных этапов [5, 36]. Сначала необходимо распознать структуру программы, а затем выдать эквивалентную программу на машинном языке. Эти два этапа — *анализ* и *синтез*. После того, как *анализатор* распознает все конструкции программы, он может установить, каким должен быть результат действия этой программы. Затем *синтезатор* вырабатывает соответствующий объектный код.

Этап анализа составляют две фазы. Первая фаза — *лексический* анализ. Поскольку исходный текст программы представляет собой последовательность символов, необходимо выделять из этой последовательности так называемые *лексемы*, то есть значимые единицы языка программирования, такие как зарезервированные слова, идентификаторы, константы, знаки операций и т.п. Для выделения лексем используется *сканер*, задачами которого являются:

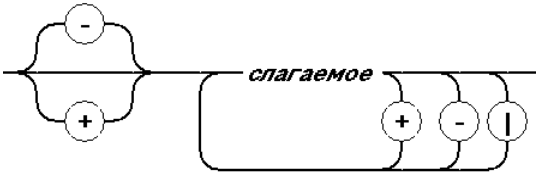
- пропуск разделителей;
- распознавание и обработка зарезервированных слов;
- распознавание и обработка идентификаторов;
- распознавание последовательности цифр в качестве чисел;
- распознавание знаков операций, скобок и т.п.

Второй фазой является *синтаксический анализ*, в нашем случае это построение дерева разбора. Разумеется, анализатор должен «знать» синтаксис языка. Синтаксис языка программирования наглядно представим в виде *синтаксических диаграмм*.

На этапе синтеза производится обход дерева и формирование объектного кода. Построение компилятора невозможно без рассмотрения формальных грамматик, ограничений на язык программирования, построения таблицы идентификаторов и пр. Но в любом языке программирования присутствуют арифметические выражения, и, не вдаваясь в подробности, на уровне здравого смысла рассмотрим синтаксис выражения. На рисунке 4.8 представлены синтаксические диаграммы простого выражения, слагаемого и множителя.

Синтаксические диаграммы отражают принятый приоритет выполнения операций: сначала вычисляем множители, затем слагаемые и наконец выражение. Из представленных диаграмм видно, что определения рекурсивны: *выражение* определяется через *множитель*, а *множитель* в свою очередь — через *выражение*. Из диаграммы *выражение* также понятно, что допускается использование унарных операций.

Простое выражение:



Слагаемое:

Множитель:

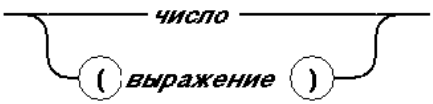


Рис. 4.8. Синтаксические диаграммы арифметического выражения

Итак, нашей целью является создание интерпретатора: сначала выражение переведем в промежуточное представление — в дерево, а затем это дерево будем обходить, вычисляя значение выражения. Для упрощения работы, не связанной с построением дерева разбора, в качестве операндов будем рассматривать только числа.

Для распознавания выражения нам потребуются следующие процедуры:

- функция *Peek()* (сканер), выделяющая очередную лексему и формирующую ее значение. Если лексема — число, то сканер формирует и последовательность цифр, соответствующих этому числу. К строке-числу сканер добавляет и унарный минус;
- функция *Expression* (выражение), распознающая выражение как арифметическую сумму слагаемых;
- функция *Term* (слагаемое), распознающая слагаемое как произведение множителей;
- функция *Factor* (множитель), которая обрабатывает числа, скобки и выявляет синтаксические ошибки.

На рисунке 4.9 представлена форма, соответствующая проекту, который создает, рисует и обходит дерево разбора, вычисляя значение выражения.

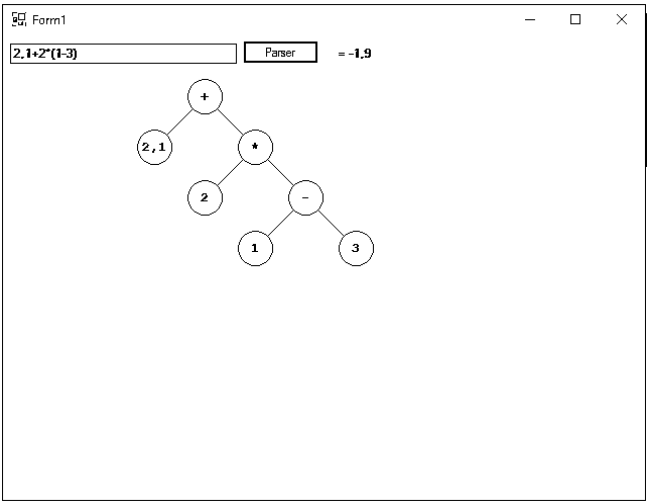


Рис. 4.9. Дерево разбора арифметического выражения

Вершины дерева и поддеревьев содержат операции, а в листьях находятся операнды. При обработке дерева, т. е. при обходе с целью вычисления выражения, операция деления выполняется как вещественное деление с округлением.

На форме размещены следующие элементы управления:

- два текстовых поля для ввода исходного выражения и для вывода результата вычисления его значения;

– кнопка «Выполнить», при нажатии на которую обрабатывается выражение с одновременным построением соответствующего дерева разбора.

Если выражение синтаксически правильное, то дерево рисуется и затем производится обход дерева для вычисления выражения и вывода результата. Узел дерева описывается следующей структурой:

Листинг 4.19. Абстрактный класс узла

```
public abstract class Node  // узел
{
   protected TypeVal typeVal;
   public abstract object DoOperation();
   protected object value;
   public object Value
   {
      get
      {
          return DoOperation();
      }
   }
}
```

В этом классе введено защищенное поле тип значения *typeVal*, свойство значение *Value* и абстрактный метод *DoOperation()*.

У класса *Node* объявлено два потомка. Первый из них – класс констант:

Листинг 4.20. Класс констант

```
public class NodeConst : Node  // узел CONST
{
    public NodeConst(TypeVal typeVal, object value)
    public override object DoOperation()
    {
```

```
            return value;
        }
    }
```

Этот класс реализует свой конструктор *NodeConst()* и перекрывает метод *DoOperation()*. Второй потомок – класс бинарных операций:

Листинг 4.21. Класс бинарных операций

```
public class NodeOperation : Node
{
    public Operation typeOperation;
    public Node left;
    public Node right;
    public NodeOperation(Operation t,
        Node left, Node right)
    public override object DoOperation()
    {
     double a=Convert.ToDouble(left.DoOperation());
     double b=Convert.ToDouble(right.DoOperation());
     switch (typeOperation)
     {
         case Operation.PLUS:
           value = a + b;
           return value;
         case Operation.MINUS:
           value = a - b;
           return value;
         case Operation.MULT:
           value = a * b;
           return value;
         case Operation.DIV:
           value = a / b;
           return value;
```

```
                default:
                    throw new NotImplementedException();
            }
        }
    }
```

В этом классе объявлено поле тип операции *typeOperation*, два поля *left* и *right* для ссылок на левое и правое поддеревья, реализован свой конструктор и перекрыт метод *DoOperation()*, который рекурсивно обращается к своим потомкам для вычисления своего значения.

После построения дерева для выражения запустить механизм вычисления выражения очень просто – надо обратиться к свойству *Value* корня дерева, а тот с помощью метода *DoOperation()* опросит всех своих потомков и вычислит у всех значения *Value*.

Выражение обрабатывается следующим образом. Сканер выделяет очередную лексему. Лексема анализируется, и в зависимости от ее значения и от текущей структуры дерева к дереву добавляется необходимое количество узлов. Дерево *надстраивается*, т. е. создается новая вершина, в двух случаях.

Во-первых, когда появляется новое слагаемое на первом уровне метода *Expression()*. На рисунке 4.10 показано, как надстроилось дерево с рисунка 4.9 при добавлении к предыдущему выражению слагаемого. Появился новый корень, а старое дерево стало левым поддеревом нового.

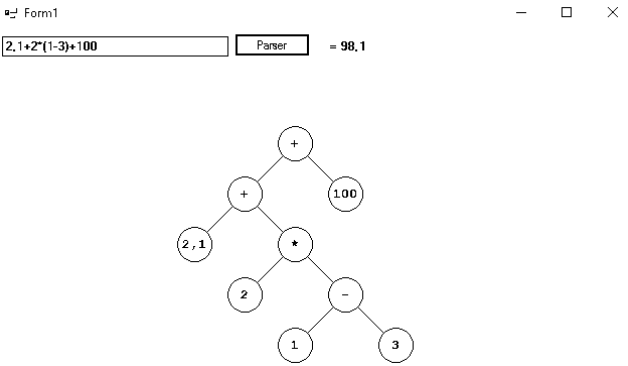
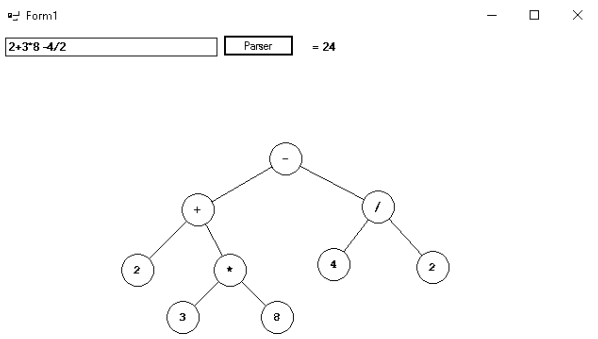
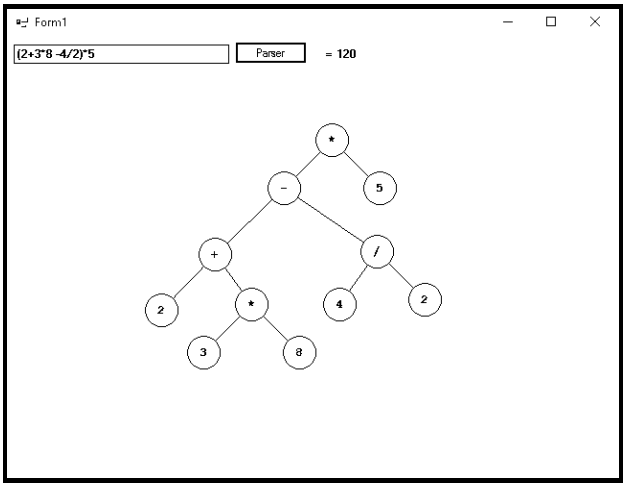


Рис.4.10. Надстроенное дерево

Второй случай надстройки дерева встречается тогда, когда в первом слагаемом выражения накапливаются множители. На рисунке 4.11*а* показано дерево, соответствующее одному множителю. Добавление множителя вызывает появление нового корня, а старое дерево, как и в предыдущем случае, становится левым поддеревом нового.



а



б

Рис. 4.11. Перестройки дерева разбора при добавлении множителя

Для обработки вложенных выражений необходимо сохранить указатель на узел, с которого начнется построение поддерева вложенного выражения, а затем, закончив обработку вложенности, вернуться к этому узлу. Для этого используются локальные переменные метода *Expression()*. Модуль *Parser*, обрабатывающий выражение и строящий дерево разбора, приведен в листинге 4.22.

Листинг 4.22. Метод Expression()

```
public void Expression(ref string s, out Node D)
{
    s = s.Trim();
    if (s.Length != 0)
    {
     string sign = "+";
     if ((s[0] == '+') | (s[0] == '-'))
         sign = Pop(ref s, 1);
     Term(ref s, out D);
     if (error != 0) return;
     s = s.Trim();
     // + - |
     while ((s.Length > 0) &&
         ((s[0] == '+') | (s[0] == '-')))
     {
         sign = Pop(ref s, 1);
         if (s.Length == 0)
           { error = 5; return; }
         Node D2;
         Term(ref s, out D2);
         if (error != 0) return;
         Node D1 = D;
```

```
                Operation t = Operation.PLUS;
                switch (sign[0])
                {
                  case '+':
                      t = Operation.PLUS; break;
                  case '-':
                      t = Operation.MINUS; break;
                }
                D = new NodeOperation(t, D1, D2);
             }
           }
           else
           {
                  D = null; error = 4;
           }
      }
```

Алгоритм метода *Expression()* в виде синтаксической диаграммы представлен на рисунке 4.12.

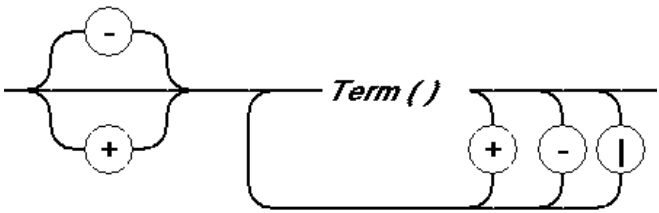


Рис. 4.12. Алгоритм метода *Expression()*

Метод *Expression()* вызывает метод *Term()*, производящий разбор слагаемых.

Листинг 4.23. Метод разбора слагаемых

```
void Term(ref string s, out Node D)
{
    s = s.Trim();
    if (s.Length != 0)
    {
     Factor(ref s, out D);
     if (error != 0) return;
     // * / &
     while ((s.Length != 0) &&
         ((s[0] == '*') | (s[0] == '/')))
     {
         string znak = Pop(ref s, 1);
         if (s.Length == 0)
           { error = 6; return; };
         Node D2;
         Factor(ref s, out D2);
         if (error != 0) return;
         Node D1 = D;

         Operation t = Operation.MULT;
         switch (znak)
         {
           case "*": t = Operation.MULT; break;
           case "/": t = Operation.DIV; break;
         }
         D = new NodeOperation(t, D1, D2);
     }
```

```
        }
        else
        {
                D = null; error = 5;
        }
    }  // Term
```

Алгоритм метода *Term()* представлен в виде синтаксической диаграммы на рисунке 4.13.

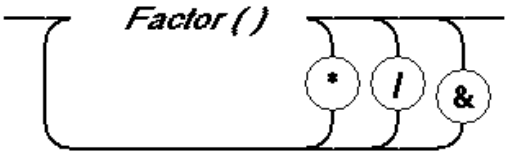


Рис. 4.13. Синтаксическая диаграмма для метода *Term()*

Метод *Term()*, в свою очередь, вызывает метод *Factor*, производящий разбор множителей и распознающий константы и выражения в скобках.

Листинг 4.24. Метод разбора множителей

```
void Factor(ref string s, out Node D)
{
    D = null;
    s = s.Trim();
    if (s.Length != 0)
    {
     if (Test(s[0], '0', '1', '2', '3', '4', '5',
          '6', '7', '8', '9', '+', '-'))
     {
         double xt; TypeVal t; int k;
```

```
            SetConst(ref s, out xt, out k, out t);
            if (error != 0) return;
            if (t == TypeVal.tFloat)
              D = new NodeConst(t, xt);
            else
              D = new NodeConst(t, k);
         }
         else
            if (s[0] == '(')  // формула в скобках
            {
              Pop(ref s, 1);
              Expression(ref s, out D);
              if (error != 0) return;
              s = s.Trim();
              if ((s.Length > 0) & (s[0] != ')'))
                  { error = 9; return; }// нет )
              Pop(ref s, 1);
        }
        }
        else
        {
           D = null; error = 6; // ожидается множитель
        }
    }
```

Метод *Factor()* замыкает рекурсию, вызывая метод *Expression()*. На рисунке 4.14 представлена диаграмма этого метода.

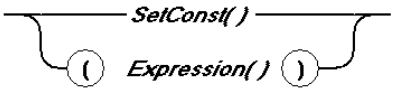


Рис. 4.14. Диаграмма метода Factor()

Синтаксическая диаграмма числовой константы представлена на рисунке 4.15.

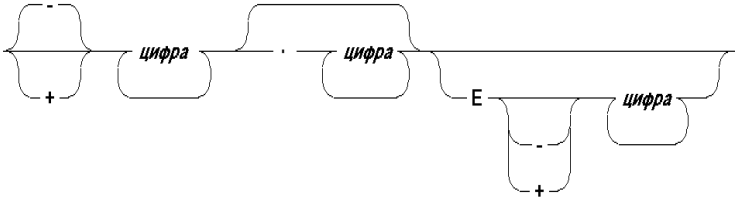


Рис. 4.15. Синтаксическая диаграмма числовой константы

Метод разбора константы представлен в листинге 4.25.

Листинг 4.25. Разбор константы

```
void SetConst(ref string s, out double x, out int k,
              out  TypeVal t)
{
    s = s.Trim(); x = 0; k = 0;
    t = TypeVal.tInt; string st = ""; x = 0;
    while(true)
    {
    if ((s != "") && Test(s[0], '0', '1', '2',
       '3', '4', '5', '6', '7', '8', '9', ','))
    {
        if (s[0] == ',') t = TypeVal.tFloat;
        st += Pop(ref s, 1);
    }
    else
        if ((s!="")&& Test(s[0], 'e', 'E') &&
```

```
                Test(s[1],

                  '0','1','2','3','4','5','6','7','8','9',
                  ',', '+','-'))
              {
                  st += s[0];
                  st += Pop(ref s, 1);
              }
          else
              break;
          } // end while(true)
          if (t == TypeVal.tInt)
              k = Convert.ToInt32(st);
          else
              x = Convert.ToDouble(st);
          s = s.Trim();
      }
```

Для запуска механизма вычисления выражения надо обратиться к свойству *Value* корня дерева, который с помощью метода *DoOperation()* опросит всех своих потомков и вычислит у всех значения *Value*.

4.5. Пример интерпретатора

Рассмотрим следующую задачу: в виде строки задано выражение, то есть конструкция из переменных, констант, бинарных операций и функций. Зная значения всех переменных, необходимо вычислить значение выражения. Бинарные операции обладают определенным приоритетом. Это означает, например, что операции умножения и деления должны выполняться раньше, чем операции сложения и вычитания. Для изменения порядка выполнения операций в выражении могут использоваться скобки. При этом любое выражение можно представить в виде дерева. Например, выражение

$$x^2 + (a-1)\cdot \ln(bx-3)$$

будет представлено деревом, содержащим 7 узлов.

На практике такая задача возникает достаточно часто. Так, например, можно создать приложение, которое будет рисовать график функции, введенной в виде строки (рис. 4.16).

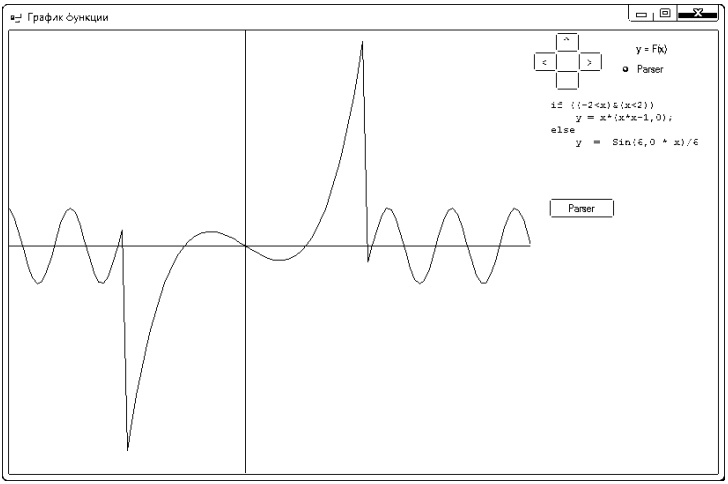


Рис. 4.16. График функции, заданной в виде строки

Далее мы будем рассматривать несколько усложненную задачу, в которой строка ввода может быть не только выражением (оператором-выражением), но и условным оператором.

Для описания типа оператора будем использовать перечислимый тип:

```
public enum OperatorType { expression, ifthen };
```

и рекурсивную структуру, определяющую операторы (листинг 4.26).

Листинг 4.26. Классы для операторов

```
public abstract class Operator
{
    public Node top;
    public OperatorType operatorType;
    public abstract void Run_Formula();
}

public class OperatorExpression : Operator
```

```
{
    public int numVal;
    public OperatorExpression(int numVal, Node t)
    public override void Run_Formula()
}

public class OperatorIf : Operator
{
    public Operator OperThen;
    public Operator OperElse;
    public OperatorIf(Node t,Operator op1,
        Operator op2)
    public override void Run_Formula()
}
```

Диаграмма этих классов представлена на рис. 4.17.

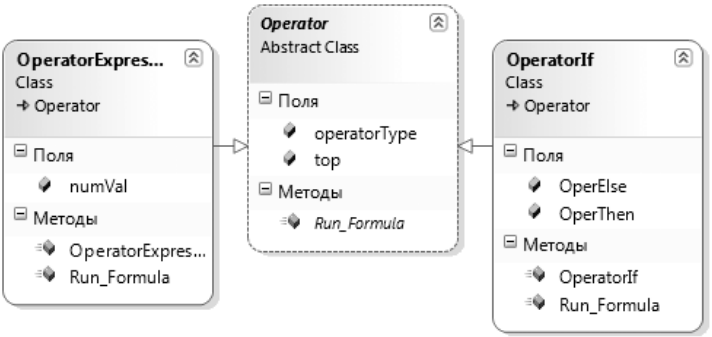


Рис. 4.17. Диаграмма классов для операторов

Дополнительно введем несколько ограничений на вводимые выражения.

1. Оператор-выражение должен иметь вид:

   <переменная> = <арифметическое_выражение>

2. Любая функция, входящая в выражение, может иметь только один параметр-значение.

3. Имена переменных могут состоять только из символов 'a'..'z', 'A'..'Z', '_', '0'..'9'.
4. Типы переменных, констант, функций и выражений могут быть или вещественными (*double*), или байт (*byte*), или логическими (*bool*), или целыми (*int*).

Для описания типов данных введем перечислимый тип:

```
public enum TypeVal { tByte, tFloat, tBool, tInt }
```

5. Каждый узел дерева выражения может быть или бинарной (унарной) операцией, или функцией, или переменной, или неименованной константой.

Для описания типа узлов введем перечислимый тип:

```
public enum TypeNode {VAL, CONST, FUNC, OPERATION }.
```

Для описания узлов дерева введем пять классов: абстрактный класс *Node* и четыре его потомка, отвечающих за константы, переменные, функции и операции.

Листинг 4.27. Создание и обход дерева разбора арифметического выражения

```
public abstract class Node  // узел
{
    protected TypeVal typeVal;
    public abstract object DoOperation();
    protected object value;
    public object Value {
get { return DoOperation(); }
    }
};

public class NodeConst : Node  // узел CONST
{
    public NodeConst(TypeVal typeVal, object value)
    public override object DoOperation()
```

```
    {
return value;
    }
}
public class NodeVal : Node
{
    protected int numVal;
    public NodeVal(int numVal)
    public override object DoOperation()
    { return aVar[numVal].value; }
}
public class NodeOperation : Node
{
    protected Operation typeOperation;
    protected Node left;
    protected Node right;
    public NodeOperation(Operation t,
Node left, Node right)
    public override object DoOperation()
}
public class NodeFunc : Node
{
    protected int numFunc;
    protected Node arg;
    public NodeFunc(int numFunc, Node arg)
    public override object DoOperation()
}
```

На рисунке 4.18 представлена диаграмма этих классов.

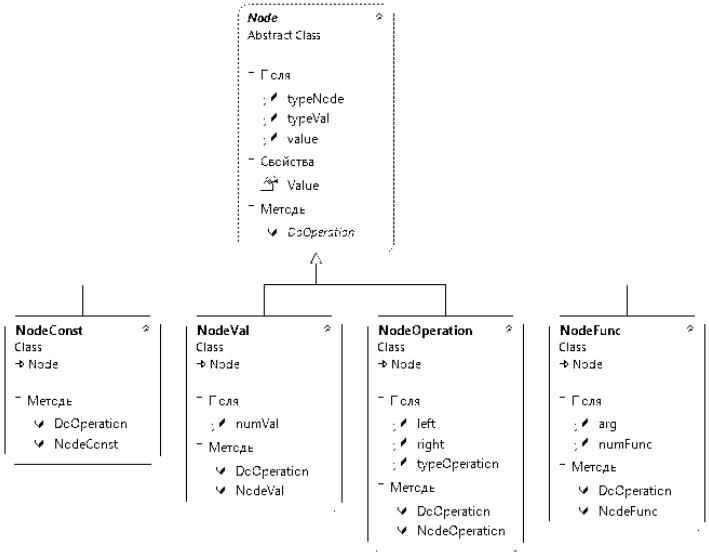


Рис. 4.18. Диаграмма классов, предназначенная для описания узлов

В этой структуре поле *value* предназначено для хранения значения узла, которое мы будем вычислять.

Если тип узла — операция *typeNode* = *OPERATION*, то поле *typeOperation* указывает на бинарную операцию, а поля *left*, *right* указывают на левый и правый операнды.

Если тип узла — функция *typeNode* = *FUNC*, то поле *numFunc* показывает на номер функции, а поле *arg* является указателем на вершину дерева, в котором сохраняется структура единственного параметра-значения функции.

Если тип узла — константа *typeNode* = *CONST*, то поле предка *value* содержит значение вещественной константы.

Если тип узла — переменная *typeNode* = *VAL*, то поле *numVal* содержит номер переменной.

Имена и значения переменных, находящихся в листьях дерева выражения будем хранить в динамическом массиве:

```
public static Tvar[] aVar = new Tvar[2];
```

Тип *TVar* содержит два поля: имя переменной *name* и значение переменной *value*:

```
public struct Tvar
{
```

```
    public string name;
    public double value;
}
```

При разборе строки могут возникать ошибки, связанные с неверной записью выражения. В этом случае переменной *Error* будем присваивать ненулевой код ошибки и аварийно прерывать разбор строки вызовом оператора *Exit*. Список сообщений, выдаваемых в случае возникновения ошибки, собран в массиве *MsgEr*:

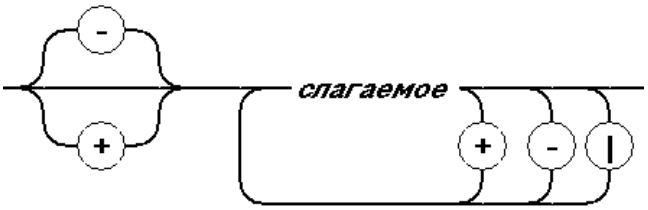
```
MsgEr: array[1..12] of string = (
    'Не найден оператор присваивания :=',
    'Не найдено THEN',
    'Ожидается выражение',
    'Ожидается простое выражение',
    'Ожидается слагаемое',
    'Ожидается множитель',
    'Неизвестная переменная',
    'Нет скобки "("',
    'Нет скобки ")"',
    'Деление на 0',
    'Ln от отрицательного числа',
    'Корень отрицательного числа'
);
```

В соответствии с усложненной постановкой задачи изменились и синтаксические диаграммы. На рисунке 4.19 представлены синтаксические диаграммы добавленного выражения, простого выражения, слагаемого и измененного множителя.

Решение задачи состоит из двух этапов: на первом этапе рекурсивной функцией *SetOperator()* создается структура данных *Operator*; на втором этапе при заданном значении *x* эта структура используется для вычисления переменной *y*.

Выражение:

Простое выражение:



Слагаемое:

Множитель:

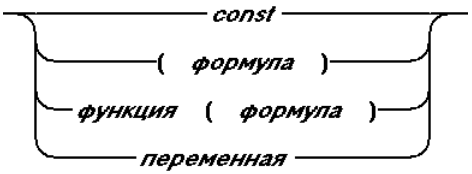


Рис. 4.19. Синтаксические диаграммы выражения

Первый этап состоит из следующих шагов.

1. С помощью процедуры *SetOperator()* определить оператор присваивания или `if`.
2. С помощью процедуры *PFormula()* проанализировать операции отношения '<', '>', '==', '!=', '<=', '>='.
3. С помощью процедуры *Expression()* проанализировать операции сложения и вычитания.

4. С помощью процедуры *Term()* проанализировать операции умножения и сложения.
5. С помощью процедуры *Factor()* проанализировать константы, скобки, функции и переменные.
6. С помощью процедуры *SetConst()* проанализировать константу.

Будем рассматривать строку, содержащую оператор, как стек, из начала которого поочередно извлекаются символы. Пробелы, естественно, игнорируются, и для изъятия пробелов используется функция *Trim():*

```
s = s.Trim();
```

Для извлечения из строки *n* символов предназначена функция *Pop()* (листинг 4.28).

Листинг 4.28. Извлечение из строки n символов

```
string Pop(ref string s, byte n)
{
   string result = s.Substring(0,n);
   s = s.Substring(n);
   s = s.Trim();
   return result;
} // взять n символов
```

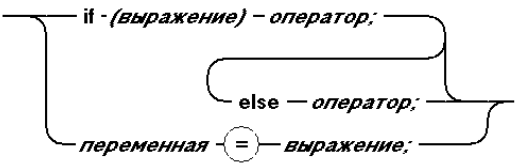


Рис. 4.20. Синтаксическая диаграмма для операторов

При разборе строки мы будем придерживаться следующего порядка. При помощи рекурсивной функции *SetOperator()* (листинг 4.29), которая возвращает код ошибки, определим тип оператора и построим узел оператора. Рассмотрим работу этой функции более подробно:

1. В строку *st* с помощью функции *TestCh()* собираем символы множества *chars*. Если *st = «if»*, то мы имеем дело с оператором *if*, иначе — с выражением.
2. Если *st = «if»*, то в строку *sExpres* собираются символы логического выражения, стоящие за *if*, а для этого собираем символы от первой скобки (до последней закрывающей скобки).
3. Вызываем процедуру *PFormula*(*sExpres*, *Op.Node)* (листинг 4.30) для создания дерева логического выражения.
4. В строку *sThen* собираются символы оператора от начала строки до *else* (если *else* в операторе присутствует). Этот оператор выполняется, если выражение — истинно.
5. Создается узел оператора *op1* типа *typeOperator* = *ifthen*.
6. Рекурсивно вызывается функция *SetOperator(ref sThen, out op1)*.
7. Если *sElse = «»*, то *op.operElse = null*, иначе рекурсивно вызывается функция *SetOperator(ref sThen, out op2)*.
8. Если *st != «if»*, то определяем номер переменной, соответствующий этой строке.
9. Создается узел оператора *op* типа *typeOperator = expression*.
10. Вызываем процедуру *PFormula(ref s, out D)* для создания дерева, соответствующего выражению в правой части оператора присваивания.

Листинг 4.29 содержит полный текст функции *SetOperator()*.

Листинг 4.29. Рекурсивная функция построения узла оператора

```
public byte SetOperator(ref string s,
    out Operator op)
{
    error = 0; s = s.Trim(); op = null;
    string st = Pop(ref s, 1);  //
    while ((s.Length > 0) && TestCh(s[0]))
           st += Pop(ref s, 1);
    s = s.Trim();
```

```
if (st == "if")  // if
{
 string sExpres = "";
 int k = 1; int i = 0;
 while ((i < s.Length) && (k != 0))
 {
     sExpres += s[++i];
     switch (s[i])
     {
       case '(': ++k; break;
       case ')': --k; break;
     }
 }
 sExpres =
    sExpres.Substring(0, sExpres.Length - 1);
 s = s.Substring(i + 1, s.Length - i - 1);
 s = s.Trim();  //

 int kThen = s.IndexOf(';');
 int kElse = s.IndexOf("else");
 if (kThen==0)
     return 2; // not THEN !
 Node D;
 PFormula(ref sExpres, out D);
 if (error!=0) return error;

 Operator op1;
 Operator op2;
 if (kElse==0)
```

```
 {
     string sThen =
       s.Substring(kThen+4,s.Length-kThen-2);
     error = SetOperator(ref sThen, out op1);
     if (error!=0) return error;
     op2 = null;
 }
 else
 {
     string sThen = s.Substring(0,kThen);
     string sElse =
       s.Substring(kElse+4,s.Length-kElse-4);
     error = SetOperator(ref sThen, out op1);
     if (error!=0) return error;
     error = SetOperator(ref sElse, out op2);
     if (error!=0) return error;
 }
 op = new OperatorIf(D,op1,op2);
}
else // переменная
{
 // поиск переменной
 bool ok = false; int nv = -1;
 while ((nv < aVar.Length - 1) && !ok)
    ok = st == aVar[++nv].name;
 if (ok)
 {
     if ((s.Length<1) || (s[0]!='='))
     {
```

```
            error = 1; return error; // not =
          }
          Pop(ref s,1);
          Node D;
          PFormula(ref s, out D);
          op = new OperatorExpression(nv,D);
        }
       }
       return error;
}
```

Для разбора операций отношения предназначен метод *PFormula():*

Листинг 4.30. Метод разбора операций отношения

```
void PFormula(ref string s, out Node D)
{
   s = s.Trim();
   if (s.Length != 0)
   {
       Expression(ref s, out D);
       if (error != 0) return;
       s = s.Trim();
       Operation chRelation = SetChRelation(s);
       if (chRelation != Operation.NONE)
       {
          switch (chRelation)
          {
             case Operation.LESS:
             case Operation.MORE:
                Pop(ref s, 1); break;
```

```
                case Operation.EQUALLY:
                case Operation.NOTEQUALLY:
                case Operation.LESSEQUALLY:
                case Operation.MOREEQUALLY:
                    Pop(ref s, 2);
                    break;
                }
                Node D2;
                Expression(ref s, out D2);
                if (error != 0) return;
                Node D1 = D;
                D = new NodeOperation(chRelation, D1, D2);
            }
        }
        else
        {
            D = null; error = 3;
        }
    }
```

Существенно изменился метод для разбора множителей, синтаксическая диаграмма которого представлена на рисунке 4.21.

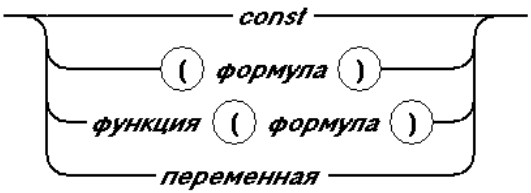


Рис. 4.21. Синтаксическая диаграмма для множителей

В метод *Factor()* добавлен разбор функций и переменных:

Листинг 4.31. Метод разбора операций отношения

```
void Factor(ref string s, out Node D)
{
   D = null;
   s = s.Trim();
   if (s.Length != 0)
   {
      char[] aCh = { '0', '1', '2', '3', '4', '5',
          '6', '7', '8', '9', '+', '-' };
      int k = Array.IndexOf(aCh, s[0]);
      if (k != -1)  // const
      {
          double x; TypeVal t;
          SetConst(ref s, out x, out k, out t);
          if (t == TypeVal.tFloat)
             D = new NodeConst(t, x);
          else
             D = new NodeConst(t, k);
      }
      else
         if (s[0] == '(')
         {// формула в скобках
            Pop(ref s, 1);
            PFormula(ref s, out D);
            if (error != 0) return;
            s = s.Trim();
            if ((s.Length > 0) && (s[0] != ')'))
                { error = 9; return; }// нет )
```

```
   Pop(ref s, 1);
}
else          // NOT | функция | переменная
{
    string st = Pop(ref s, 1);
    // стандартная функция | переменная
    while ((s.Length > 0) && TestCh(s[0]))
       st += Pop(ref s, 1);
    bool ok = false; int nf = -1;
    // поиск функции
    while ((nf<nameFunc.Length - 1) && !ok)
       ok = st == nameFunc[++nf];
    if (ok)
    {
       if ((s.Length == 1)||(s[0] != '('))
       {// нет скобки "("
          error = 8; return;
       }
       s = s.Trim();
       st = ""; k = 1; int i = 0;
       while ((i < s.Length) && (k != 0))
       {
          st += s[++i];
          switch (s[i])
          {
             case '(': ++k; break;
             case ')': --k; break;
          }
       }
```

```
                    st = st.Substring(0, st.Length - 1);
                    s = s.Substring(i, s.Length-i-1);
                    Node Da;
                    PFormula(ref st, out Da);
                    if (error != 0) return;
                    D = new NodeFunc(nf, Da);
                    s = s.Trim();
                    return;
                 }
                 // поиск переменной
                 ok = false; int nv = -1;
                 while ((nv < aVar.Length - 1) && !ok)
                    ok = st == aVar[++nv].name;
                if (ok)
                {
                   D = new NodeVal(nv);
                   s = s.Trim(); return;
                }
                error = 7;   // неизвестная переменная
             }
          }
          else
          {
             D = null; error = 6; // ожидается множитель
          }
       }
```

Вычисление переменных на втором этапе решения реализуется с помощью метода *Run_Formula()* класса *Operator*:

Листинг 4.32. Первый вызов метода `Run_Formula()`

```
private double F(double x)
{
   TParser.aVar[0].value = x;
   parser.topOp.Run_Formula();
   return TParser.aVar[1].value;}
```

У класса абстрактного класса *`Operator`* есть метод *`Run_Formula()`* (листинг 4.33), который перекрывают потомки.

Листинг 4.33. Метод `Run_Formula()` класса `Operator`

```
public abstract class Operator
{
   public Node top;
   public OperatorType operatorType;
   public abstract void Run_Formula();
}
```

В классе оператора-выражения перекрытый метод записывает значение, взятое из вершины дерева разбора правой части, в переменную левой части:

Листинг 4.34. Метод `Run_Formula()` класса `OperatorExpression`

```
protected class OperatorExpression : Operator
{
    public override void Run_Formula()
    {
       aVar[numVal].value =
           Convert.ToDouble(top.Value);
    }
}
```

Класс оператора *if* вычисляет значение логического условия и рекурсивно вызывает метод *Run_Formula()* для своих ссылок *OperThen* и *OperElse*:

Листинг 4.35. Метод `Run_Formula()` класса `OperatorIf`

```
protected class OperatorIf : Operator
{
   public override void Run_Formula()
   {
      if (Convert.ToBoolean(top.Value))
         OperThen.Run_Formula();
      else
         if (OperElse != null)
            OperElse.Run_Formula();
   }}
```

Выводы

Деревья – это структуры данных, предназначенные для представления иерархически организованных данных. Каждый узел дерева является объектом, содержащим поля данных и поля-ссылки для связи с узлами-потомками. В бинарных деревьях каждый узел имеет не более двух потомков. Упорядоченные бинарные деревья формируются по алгоритму, обеспечивающему в дальнейшем быстрый поиск с эффективностью $O(log_2N)$, где N – число узлов.

При необходимости просмотра всех узлов двоичного дерева, например, для печати хранящихся в нем значений используются рекурсивные алгоритмы обхода с различным порядком посещения узлов (*inOder, preOder*, *postOder*). Для более глубокого знакомства с алгоритмами обработки деревьев можно рекомендовать монографии [12 – 14, 23].

Упражнения

1. Создать приложение, в котором генерируется последовательность из 1000 целых чисел в диапазоне [0, 99], сохраняемых в двоичном дереве поиска. Выполнить обход дерева и сформировать массив, каждый

элемент которого содержит кратность вхождения в последовательность числа, равного индексу этого элемента.

2. В качестве варианта задания 1 рассмотреть возможность сохранения в узлах дерева отдельных слов из заданного текстового файла.
3. Добавить в число методов класса `BinarySearchTree` функцию, подсчитывающую количество дуг в дереве.
4. Измените класс `BinarySearchTree` так, чтобы его можно было использовать для вычисления значений бесскобочных арифметических выражений (например, 2 + 3 * 4 / 5).
5. Используйте двоичное дерево для представления файловой системы: каждый каталог содержит не более двух наследников – файлов и/или папок. Для файла, кроме имени, в виде целого числа указано время его содания. Выполнить последующий обход дерева с удалением всех файлов, созданных ранее указанной даты.

Глава 5. Графы

Структуры данных можно классифицировать, исходя из характера отношений между их элементами. Так, элементы списков находятся в *отношении следования*: «предыдущий» – «последующий». Для деревьев это *отношение иерархии*: «родитель» – «потомок». В графах таким отношением является *отношение связности.* Как и элементы дерева, элементы графа называются *узлами,* а связь между двумя узлами называется *ребром графа.* Ниже в этой главе будут приведены строгие определения понятий теории графов и рассмотрен ряд важных теорем.

В настоящее время теория графов является важным разделом дискретной математики, но зародилась она в связи с решением развлекательных математических задач и головоломок. Общепринято считать, что начало этому положил Леонард Эйлер, сформулировавший в 1736 году знаменитую задачу о семи кенигсбергских мостах. Другими известными примерами подобных задач являются задача о четырех красках (Ф. Гутри, 1852 г.) и задача коммивояжера (К. Менгер, 1830 г.).

Бурное развиттие теории графов в середине XIX века было связано с использований ее практических приложений в различных областях естествознания. Многие результаты, относящиеся к теории графов, были получены при решении практических проблем. Немецкий физик Густав Кирхгоф использовал графы для представления электрических схем и анализа полной системы уравнений для токов и напряжений. В основе такого анализа лежал поиск в графе подструктур, представляющих собой линейно независимые контуры. Английский математик XIX века Артур Кэли, решавший задачу подсчета числа изомеров предельных углеводородов, свел ее к подсчету числа деревьев, обладающих заданными свойствами, которые могут быть выделены в графе. Позже методы теории графов стали использоваться также и в других раделах математики, с чем было связано получение важных результатов в алгебре, теории вероятностей, топологии, теории чисел.

Среди задач теории графов можно выделить задачи, носящие преимущественно комбинаторный или преимущественно геометрический характер. К первому классу относятся, например, задачи о подсчете и перечислении графов с заданными свойствами. Ко второму – задачи, связанные с обходами графа, а также задачи об укладке графа на различных поверхностях. Важной проблемой, специфической для теории графов, является изучение различных свойств связности графов и их изоморфизма. Результаты таких исследований находят свое применение при анализе надежности электронных схем и коммуникационных сетей. Еще одним направлением исследований в теории гра-

фов является цикл проблем, связанных с разбиением множества вершин (ребер) на подмножества по какому-либо признаку. При этом смежные вершины (ребра) должны принадлежать различным подмножествам. Данная задача известна как задача о раскраске графа.

5.1. Основные определения теории графов

Пусть V — непустое конечное множество. Через $V(2)$ обозначим множество всех двуэлементных подмножеств из V.

Графом G называется пара множеств (V, E), где E — произвольное подмножество из $V(2)$. Элементы множеств V и E соответственно называются *вершинами* и *ребрами* графа G. Если v_1, v_2 — вершины, а $e = (v_1, v_2)$ — соединяющее их ребро, тогда вершина v_1 и ребро e *инцидентны*, вершина v_2 и ребро e тоже *инцидентны*. Два ребра, инцидентные одной вершине, называются *смежными*; две вершины, инцидентные одному ребру, также называются *смежными*.

Граф G называется *ориентированным графом* (*орграфом*), если все его ребра являются *ориентированными.* Ориентированные ребра называют *дугами.* Дуга описывается как упорядоченная пара вершин (v, w), где вершину v называют началом, а w — концом дуги. Говорят, что дуга $v \to w$ ведет от вершины v к смежной с ней вершине w.

Планарным графом называется граф, который может быть изображен на плоскости без пересечения ребер.

Граф A называется *двойственным* для планарного графа B, если A содержит столько вершин, сколько имеется граней в B, при этом каждое ребро графа A пересекает ровно одно ребро графа B.

Полным графом называется граф, в котором для каждой пары вершин (v_1, v_2) существует ребро, инцидентное v_1 и инцидентное v_2. Иначе говоря, граф $G(V, E)$ называется *полным*, если любая пара его вершин соединена хотя бы в одном направлении.

Псевдографом называется граф, содержащий петли, а *мультиграфом* – граф, в котором существует пара вершин, соединенная более чем одним ненаправленным ребром, либо более чем двумя дугами противоположных направлений.

Вполне несвязный граф – это граф без ребер; используются другие названия: регулярный степени 0 граф, пустой граф, нуль-граф.

Двудольным называется граф, множество V вершин которого разбито на два непересекающихся подмножества V_1 и V_2, причем каждое ребро графа соединяет вершину из V_1 с вершиной из V_2. Множества V_1 и V_2 называются *долями* двудольного графа.

Упорядоченным называется граф, в котором дуги, выходящие из каждой вершины, однозначно пронумерованы, начиная с 1. Дуги считаются упорядоченными в порядке возрастания номеров. При графическом представле-

нии часто дуги считаются упорядоченными в порядке некоторого стандартного обхода (например, слева направо).

Степень или *валентность вершины* $deg(v)$ – это количество ребер, инцидентных вершине v. Минимальная степень вершины графа G обозначается $\delta(G)$, а максимальная – $\Delta(G)$.

Маршрут в графе G – это чередующаяся последовательность вершин и ребер $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$, в которой любые два соседних элемента инцидентны. Если $v_0 = v_k$, то маршрут *замкнут*, иначе *открыт*. Другими словами, *маршрутом* называется чередующаяся последовательность вершин и ребер $v_0, e_1, v_1, \ldots, v_{t-1}, e_t, v_{t+1}$, в которой $e_i = v_{i-1}v_i$ $(1 \le i \le t)$. Такой маршрут кратко называют (v_0, v_t)-*маршрутом* и говорят, что он *соединяет* v_0 с v_t; а вершины v_0, v_t – *концевые* вершины указанного маршрута.

Длина маршрута – это количество содержащихся в нем ребер с возможными повторениями. Так, для маршрута $M = (v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k)$ длина равна k и обозначается как $|M| = k$.

Цепью в графе называется маршрут, все ребра которого различны. Если при этом все вершины различны, то такая цепь называется *простой*. В цепи $v_0, e_1, \ldots, e_k, v_k$ вершины v_0 и v_k называются *концами* цепи. Цепь с концами u и v соединяет вершины u и v.

Цикл в графе — это цепь, которая начинается и заканчивается в одной и той же вершине. Цикл, в котором нет повторяющихся вершин, кроме совпадающих начальной и конечной, называется *простым* циклом. Простой цикл орграфов называется *контуром*.

Гамильтоновым циклом называется простой цикл, который проходит через все вершины графа. Граф, содержащий такой цикл, называется *гамильтоновым графом*.

Вершина называется *висячей*, если ее степень равна 1 (т. е. $deg(v) = 1$) и *изолированной,* если ее степень равна 0 (т. е. $deg(v) = 0$).

Два графа называются *изоморфными*, если существует такая перестановка вершин, при которой они совпадают.

Подграф исходного графа – это граф, содержащий некое подмножество вершин данного графа и все ребра, инцидентные данному подмножеству.

Множеством смежности вершины v называется множество вершин, смежных с вершиной v. Обозначается как $\Gamma_+(v)$.

Полустепень захода в орграфе для вершины v – это число дуг, входящих в вершину. Обозначается $deg_+(v)$.

Полустепень исхода в орграфе для вершины v – это число дуг, исходящих из вершины. Обозначается $deg_-(v)$.

Регулярным называется граф, степени всех вершин которого одинаковы. Степень регулярности является инвариантом графа G и обозначается $r(G)$. Для нерегулярных графов $r(G)$ не определено.

Раскраской графа называется разбиение множества его вершин на подмножества, называемые *цветами*, при котором нет двух смежных вер-

шин, принадлежащих одному и тому же подмножеству. *Хроматическое число* графа – это минимальное количество цветов, требуемое для раскраски графа.

Размеченный граф – это граф, для которого задано множество меток S, а также функции разметки вершин f: $A \to S$ и разметки дуг g: $R \to S$. Графически эти функции представляются надписыванием меток на вершинах и дугах. Множество меток может разделяться на два непересекающихся подмножества меток вершин A и меток дуг R.

Две вершины в графе *связаны*, если существует соединяющая их простая цепь. Граф называется *связным,* если все вершины в нем связаны.

5.2. Проект для алгоритмов на графах

Для иллюстрации алгоритмов работы с графами удобно использовать их наглядное представление. Поэтому начнем наше изложение с описания проекта, реализующего подобное представление для графов. Проект состоит из главной формы `FormGraph`, шести вспомогательных форм `FormPropertyGraph`, `FormListBox`, `FormMatr`, `FormTools`, `FormViewGraph` и модуля `LibGraph`, в котором собраны структуры данных и все алгоритмы для графов (рис. 5.1).

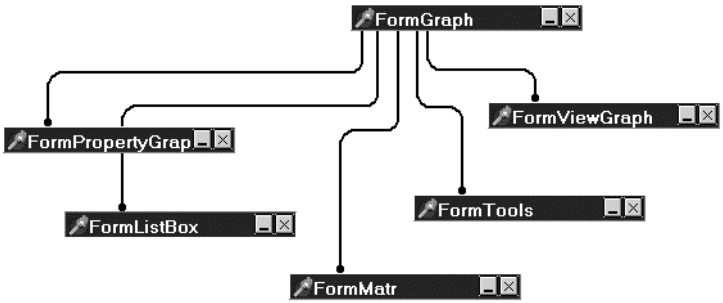


Рис. 5.1. Структура проекта

Изображение графа строится на форме `FormGraph` и выполняется в трех видах: в виде прямоугольников, соединенных ломаными линиями (рис. 5.2); в виде окружностей, соединенных прямыми линиями (рис. 5.3); в виде прямоугольников типа верхней полосы формы (рис. 5.1). Переключение типа графа происходит на форме `FormViewGraph`. На форме `FormTools` находятся три инструментальные кнопки, позволяющие добавлять и перемещать узлы, создавать дуги. Форма `FormMatr` показывает матрицу смежно-

сти, а с помощью формы *FormPropertyNode* можно задавать свойства любого узла.

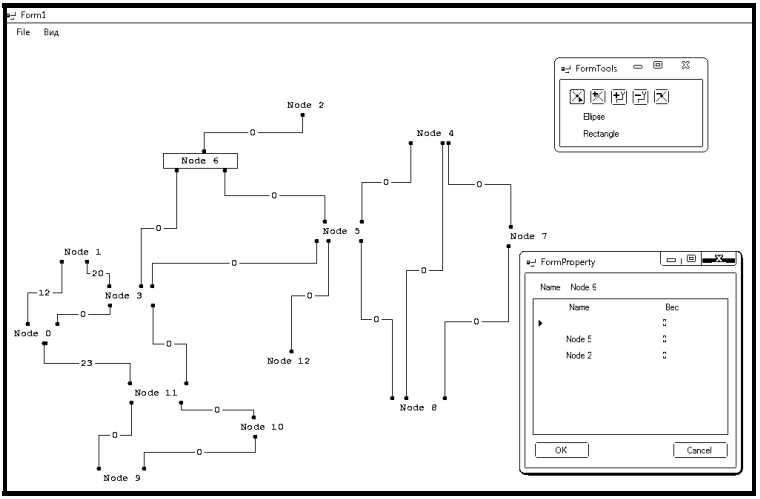


Рис. 5.2. Проект для алгоритмов на графах, узлы – прямоугольники

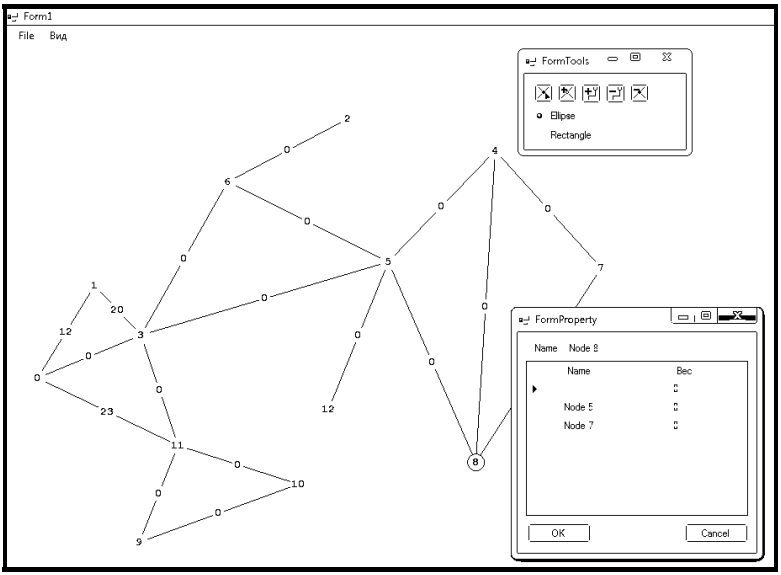


Рис. 5.3. Проект для алгоритмов на графах, узлы – окружности

5.2.1. Структура стек для обработки графов

Многие алгоритмы на графах требуют помещать данные в стек или в очередь. Напомним, что стек — это последовательность однотипных элементов, в которую можно включать новые элементы и удалять из нее элементы по принципу LIFO «последним пришел – первым вышел», т. е. первым удаляется элемент, который был добавлен последним. Очередь – это последовательность однотипных элементов, в которую можно включать новые элементы и удалять из нее элементы по принципу FIFO «первым пришел – первым вышел». Мы выберем наиболее простой механизм этих структур, основанный на массиве, объединив в одном классе и стек, и очередь (листинг 5.1).

Листинг 5.1. Структура стека/очереди

```
public class MyStack
{
   private Node top;
   private Node tail;
   public MyStack()                    // конструктор
   public void Push(object data)       // положить в
стек
   public object Pop()                 // взять из сте-
ка
   public bool isEmpty()               // проверка на
пустоту
   public class Node                   // узел
   public void PushQueue(object inf)//вставить в
хвост очереди
   public string StackToStr()
}
```

В этом классе носителем информации является экземпляр класса *Node*, у которого есть поле *data* типа *object*.

Листинг 5.2. Класс узла

```
public class Node                   // узел
{
```

```
    public Node next;
    public object data;
    public Node(Node next, object data) // конструк-
тор
    {
        this.next = next;
        this.data = data;
    }
}
```

Поля *top* и *tail* показывают на начало стека и конец очереди. Для реализации механизмов работы со стеками и очередями введем две переменные *myStack* и *path*. Для работы со стеком предназначены следующие процедуры:

- инициализация;
- проверка на пустоту стека;
- помещение элемент в стек;
- извлечение элемент из стека.

При инициализации необходимо просто направить вершину и конец стека на *null* (листинг 5.3).

Листинг 5.3. Инициализация стека/очереди

```
public MyStack()  // конструктор
{
    top = null;
    tail = null;
}
```

При проверке пустоты стека необходимо проверить, что вершина стека показывает на *null* (листинг 5.4).

Листинг 5.4. Проверка пустоты стека/очереди

```
public bool isEmpty() // проверка на пустоту
{
    return top == null;
}
```

При добавлении необходимо создать новый экземпляр узла, поле *next* которого будет показывать на вершину, и переместить вершину на новый элемент (листинг 5.5).

Листинг 5.5. Добавление элемента в стек/очередь

```
public void Push(object data)// положить в стек
{
   top = new Node(top, data);
   if (top.next == null)
      tail = top;
}
```

При извлечении элемента из вершины стека необходимо проверять его на пустоту и, если стек не пуст, то возвращать элемент и переместить вершину на следующий элемент (листинг 5.6).

Листинг 5.6. Извлечение элемента из вершины стека

```
public object Pop() // взять из стека
{
   if (top == null)
      throw new InvalidOperationException();
   object result = top.data;
   top = top.next;
   return result;
}
```

При добавлении элемента в конец списка необходимо создать новый элемент. Если список пуст, то и вершина и конец списка направляются на этот элемент, иначе поле *next* последнего элемента списка направляем на новый элемент и перемещаем указатель на конец списка на новый элемент (листинг 5.7).

Листинг 5.7. Извлечение элемента из конца списка

```
public void PushQueue(object inf)
```

```
// положить в хвост очереди
{
     Node p = new Node(null, inf);
     if (isEmpty())
     {
         top = p; tail = p;
     }
     else
     {
         tail.next = p; tail = p;
     }
}
```

5.2.2. Структура данных для представления графов

Выбор структуры данных оказывает решающее значение на эффективность алгоритмов. Общая структура классов проекта представлена на рисунке 5.4.

Класс *Graph* является основным.

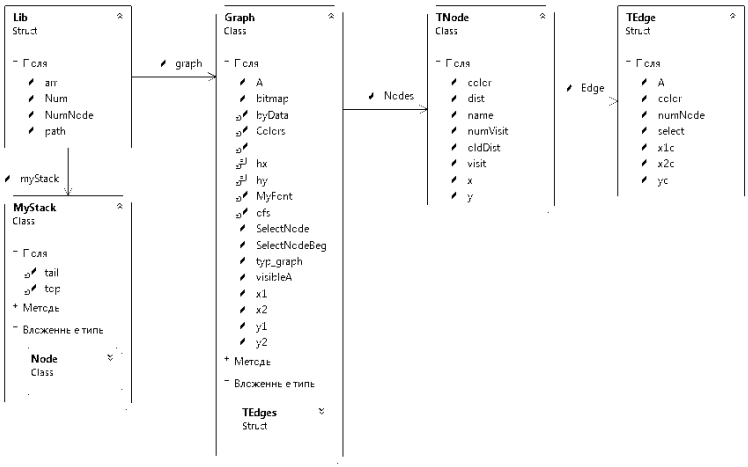


Рис. 5.4. Общая структура классов проекта

Листинг 5.8. Класс Graph

```
public class Graph
{
      const int hx = 50, hy = 10;
      public Bitmap bitmap;
      public TNode[] Nodes = new TNode[0];// узлы
      //выделенный узел
      public static TNode SelectNode;
      public static TNode SelectNodeBeg;
      public byte typ_graph = 1;
      public int[,] A;   // матрица инцидентности
      // установить граф неориентированным
      public void SetSim()
      public Graph(int VW, int VH)
      public int FindNumEdge(int i, int j)
      public void SetA()
      // добавить узел
      public void AddNode(int x,int y)
      public void AddEdge()  // добавить ребро
      // найти узел
      public TNode FindNode(int x, int y)
      public void DeSelectEdge()
      public void Draw(bool fl)   // нарисовать
      public void Save(string FileName)   // записать
      // прочитать
      public void Open(string FileName)
      // найти ребро
      public int FindLine(int x, int y,
```

```
                out int NumLine)
        // удалить ребро
        public void DelEdge(int NumNode, int NumEdge)
}
```

Основной полем класса графа при реализации алгоритмов является динамический массив узлов *Node[]* типа *TNode*. По массиву ребер, имеющемуся у каждого узла, можно построить матрицу инциденций *int[,]A*.

Каждый элемент массива *Node* вершин графа определяется структурой, представленной в листинге 5.9.

Листинг 5.9. Структура узлов графа

```
public class TNode
{
    public string name;  // имя узла
    public TEdge[] Edge; // массив дуг (ребер)
    public bool visit;   // признак "узел посещен"
    public int x0, y0;   // координаты центра узла
    public int numVisit; // № посещения
    public Color color;  // цвет узла
    public int dist;     // минимальное расстояние
}
```

В этой структуре поле *name* предназначено для хранения имени узла, поле *Edge* описывает список ребер, выходящих из вершины, поля *x0* и *y0* задают координаты центра вершины. Поле *visit* будет играть важную роль при реализации многих алгоритмов, в нем мы будет отмечать посещение вершины. Поле *color* также играет вспомогательную роль при решении задач раскраски графов, а поле *dist* будет использоваться при решении задач определения кратчайших путей на графе.

В листинге 5.10 представлена структура, предназначенная для описания ребер. Самым важным в этой структуре является поле *numNode*, содержащее номер вершины, на которую показывает ребро. Не менее важную информацию содержит поле *A*, содержащее вес ребра. Для веса ребра мы ограничились целым типом, хотя в реальных задачах это поле может быть вещественным.

Листинг 5.10. Структура ребер

```
public struct TEdge
{
   public int A;            //
   public int numNode;      //
   public int x1c, x2c, yc; //
   public Color color;
   public bool select;
}
```

Поле *color* (цвет дуги) играет вспомогательную роль: при реализации некоторых алгоритмов мы будем менять цвет ребра, если пройдем по нему. Поля *x1c*, *y1c* и *yc* содержат геометрические параметры ребра (рис. 5.5). Поле указывает на расстояние горизонтальной части ребра от верхнего края формы.

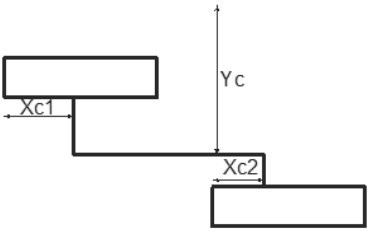


Рис. 5.5. Геометрические параметры ребра

5.2.3. Изображение графов

Функция рисования графа похожа на рисование дерева в проекте, описанном в пп. 4.2.1. На канву *bitmap* мы сначала выводим дуги, а затем все узлы (листинг 5.11).

Листинг 5.11. Рисование графа

```
public void Draw(bool fl) // нарисовать
{
   using (Graphics g = Graphics.FromImage(bitmap))
   {
```

```
Color cl = Color.FromArgb(255, 255, 255);
g.Clear(cl);
Pen MyPen = Pens.Black;
SolidBrush MyBrush =
      (SolidBrush)Brushes.White;
string s;
int N = Nodes.Length;

//Line
for (int i = 0; i < N; i++)
{
   if (Nodes[i].Edge != null)
   {
      int L = Nodes[i].Edge.Length;
      MyBrush.Color = Color.White;
      for (int j = 0; j < L; j++)
      {
        Edge = Nodes[i].Edge[j];
        switch (typ_graph)
        {
           case 0:
             if (Edge.select)
               MyPen = Pens.Red;
             else
               MyPen = new
                     Pen(Edge.color);
             int a1 = Nodes[i].x;
             int b1 = Nodes[i].y;
             int a2 =
```

```
                Nodes[Edge.numNode].x;
            int b2 =
                Nodes[Edge.numNode].y;
            g.DrawLine(MyPen,
                  new Point(a1,b1),
                Nodes[Edge.numNode].x;
            int b2 =
                Nodes[Edge.numNode].y;
            g.DrawLine(MyPen,
                  new Point(a1,b1),
                new Point(a2, b2));
            s = Convert.ToString(Edge.A);
            SizeF size =
                  g.MeasureString(s,MyFont);
            if (Lib.graph.visibleA)
            {
               g.FillRectangle(Brushes.White,
                  (a1+a2)/2 - size.Width/2,
                  (b1+b2)/2 - size.Height/2,
                  size.Width, size.Height);
               g.DrawString(s, MyFont,
                     Brushes.Black,
                  (a1+a2)/2-size.Width/2,
                  (b1+b2)/2-size.Height/2);
             }
             break;
        }
    }
}

// Nodes
for (int i=0; i<N; i++)
{
  if (Nodes[i] == SelectNode)
```

```
      MyPen = Pens.Red;
   else
      MyPen = Pens.Silver;
   if (Nodes[i].visit)
      MyBrush.Color = Color.Silver;
   else
     if (Nodes[i] == SelectNode)
       MyBrush.Color = Color.Yellow;
     else
       MyBrush.Color = Color.LightYellow;
   switch (typ_graph)
   {
     case 0:
       MyBrush.Color = Nodes[i].color;
       g.FillEllipse(MyBrush,
            Nodes[i].x - hy,
          Nodes[i].y - hy, 2 * hy, 2 * hy);
       g.DrawEllipse(Pens.Black,
            Nodes[i].x - hy,
          Nodes[i].y - hy, 2 * hy, 2 * hy);
       s = Convert.ToString(i);
       SizeF size =
           g.MeasureString(s,MyFont);
       g.DrawString(s, MyFont, Brushes.Black,
           Nodes[i].x - size.Width/2,
           Nodes[i].y - size.Height/2);
       break;
   }
   if (fl)
```

```
                g.DrawLine(MyPen, new Point(x1,y1),
                    new Point(x2,y2));
            }
        }
```

Как и раньше, для перемещения узлов и создания новых дуг реализованы обработчики событий *onMouseDown*, *onMouseMove* и *onMouseUp*.

5.2.4. Запись и чтение графов

Запись графа будем осуществлять с помощью файлового потока, который создадим с помощью класса *FileStream*. Запись состоит из следующих шагов:

1) создать экземпляр класса *FileStream*;
2) создать байтовый массив *byData[]*, в который поместятся все данные о графе;
3) заполнить массив данными из графа;
4) записать массив *byData[]*;
5) закрыть файловый поток.

Все эти шаги реализованы в методе *Save():*

Листинг 5.12. Запись данных о графе

```
public void Save(string FileName)  // записать
{
    ofs = 0;
    FileStream aFile =
new FileStream(FileName, FileMode.Create);
    int N = LengthFile();
    byData = new byte[N];

    int L1 = Nodes.Length;
    IntInData(L1);
    for (int i = 0; i <= L1 - 1; i++)
    {
IntInData(Nodes[i].x);
```

```
IntInData(Nodes[i].y);
StrInData(Nodes[i].name);
int L2 = 0;
if (Nodes[i].Edge != null)
    L2 = Nodes[i].Edge.Length;
IntInData(L2);
for (int j = 0; j <= L2 - 1; j++)
{
    IntInData(Nodes[i].Edge[j].A);
    IntInData(Nodes[i].Edge[j].x1c);
    IntInData(Nodes[i].Edge[j].x2c);
    IntInData(Nodes[i].Edge[j].yc);
    IntInData(Nodes[i].Edge[j].numNode);
}
    }
    aFile.Write(byData, 0, N);
    aFile.Close();
}
```

Для записи потребовалось три вспомогательных метода. Первый *LengthFile*() предназначен для вычисления длины байтового массива, в который поместятся все данные о массиве:

Листинг 5.13. Вычисление длины байтового массива

```
protected int LengthFile()  // вычислить размер фай-
ла
{
   int n = 4;
   int L1 = Nodes.Length;
   for (int i=0; i<=L1-1; i++)
   {
```

```
            n += 16+4*Nodes[i].name.Length;
            int L2=0;
            if (Nodes[i].Edge != null)
               L2 = Nodes[i].Edge.Length;
            n += L2 * 20;
         }
         return n;
      }
```

Второй метод перемещает переменную целого значения в байтовый массив и сдвигает смещение *ofs* на 4:

Листинг 5.14. Перемещение целого значения в байтовый массив

```
protected void IntInData(int k)
{
   byte[] byByte;
   byByte = BitConverter.GetBytes(k);
   byByte.CopyTo(byData, ofs); ofs += 4;
}
```

Для перемещения строки в байтовый массив предназначен метод *StrInData()*. Так как предполагается использование кириллицы в строках, то приходится использовать кодировку UTF-32, которая требует 4 байта на символ. Перемещение происходит в четыре этапа:

- перемещаем длину строки в основной байтовый массив *byData;*
- перемещаем строку в символьный массив *charData[];*
- с помощью класса *Encoder* перемещаем символьный массив во вспомогательный байтовый массив *byByte[]*;
- вставляем вспомогательный байтовый массив *byByte[]* в основной байтовый массив *byData*.

Листинг 5.15. Перемещение строки в байтовый массив

```
protected void StrInData(string s)
{
```

```
        byte[] byByte;
        int L = s.Length; IntInData(L);
        char[] charData = s.ToCharArray();
        byByte = new byte[4 * charData.Length];
        Encoder e = Encoding.UTF32.GetEncoder();
        e.GetBytes(charData, 0, charData.Length, byByte,
           0, true);
        byByte.CopyTo(byData, ofs); ofs += 4 * L;
    }
```

Чтение файла происходит в том же порядке (листинг 5.16):

1) создать экземпляр класса *FileStream*;
2) создать байтовый массив *byData[]*, в который поместятся все данные о графе;
3) прочитав файл, заполнить массив данными из файла;
4) пройдя по массиву *byData[]*, создать все узлы и ребра графа;
5) закрыть файловый поток.

Листинг 5.16. Чтение данных о графе

```
    public void Read(string FileName) // прочитать
    {
        ofs = 0;
        FileStream aFile =
           new FileStream(FileName, FileMode.Open);
        int N = (int)aFile.Length;
        byData = new byte[N];
        aFile.Read(byData, 0, N);
        int L1 = DataInInt();
        Nodes = new TNode[L1];
        for (int i = 0; i <= L1 - 1; i++)
        {
    Nodes[i] = new TNode();
```

```
        Nodes[i].x = DataInInt();
        Nodes[i].y = DataInInt();
        Nodes[i].name = DataInStr();
        int L2 = DataInInt();
        Nodes[i].Edge = new TEdge[L2];
        if (L2 != 0)
            for (int j = 0; j <= L2 - 1; j++ )
            {
               Nodes[i].Edge[j].A = DataInInt();
               Nodes[i].Edge[j].x1c = DataInInt();
               Nodes[i].Edge[j].x2c = DataInInt();
               Nodes[i].Edge[j].yc = DataInInt();
               Nodes[i].Edge[j].numNode = DataInInt();
               Nodes[i].Edge[j].color = Color.Silver;
            }
            }
            aFile.Close();
        }
```

Для чтения потребовался вспомогательный метод *DataInInt()*, который извлекает целое значение из массива *byData[]*:

Листинг 5.17. Извлечение целого значения из массива `byData[]`

```
        protected int DataInInt()
        {
            int result = BitConverter.ToInt32(byData, ofs);
            ofs += 4;
            return result;
        }
```

и метод извлечения строки из массива *byData[]*:

Листинг 5.18. Извлечение строки из массива `byData[]`

```
protected string DataInStr()
{
    byte[] byByte;
    int L = DataInInt();
    byByte = new byte[4 * L];
    for (int j = 0; j <= 4 * L - 1; j++)
byByte[j] = byData[j + ofs];
    char[] charData = new char[L];
    Decoder d = Encoding.UTF32.GetDecoder();
    d.GetChars(byByte, 0, byByte.Length, charData,
0);
    string s = "";
    for (int j = 0; j < charData.Length; j++)
s += charData[j];
    ofs += 4 * L;
    return s;
}
```

Извлечение строки снова происходит в четыре этапа:

- извлекаем длину строки *L*;
- заполняем вспомогательный массив *byByte[]* длиной *4*L*;
- с помощью класса *Decoder* перемещаем данные из массива *byByte[]* в символьный массив *charData[];*
- перемещаем данные из массива *charData[]* в строку.

5.3. Поиск в графах

Основой любого алгоритма поиска нужной вершины в графе является перебор вершин, организованный так, чтобы каждая вершина просматривается не более одного раза. Критериями эффективности любого алгоритма и, в частности, алгоритмов поиска являются:

- легкая читаемость алгоритма;

– выполнение требования однократности анализа любого ребра или вершины.

5.3.1. Поиск в глубину

Поставим задачу поиска кратчайшего пути между двумя вершинами неориентированного графа. Очевидно, что речь идет об алгоритме, реализующем некоторый перебор возможных путей. Основной проблемой является определение эффективного порядка перебора. Одним из подходов к решению этой задачи является так называемый *поиск в глубину* (*depth first search*). Общая идея этого метода заключается в следующем: начиная поиск с какой-то вершины *n*, просматриваем список инцидентных ей ребер и, если находится соседняя непосещенная вершина, то переходим в нее, далее продолжаем алгоритм из этой вершины. Естественно, мы должны отмечать каждую посещенную вершину, присваивая полю `Node[n].visit` значение `true`.

5.3.1.1. Алгоритм обхода графа с поиском в глубину

Пусть *G* — неориентированный связный граф. Для организации поиска будем использовать признак посещенности узла `numVisit` и признак прохождения по ребру `select`. Перед началом поиска булевские поля `select` получают значение `false` (ребра не помечены), а целочисленные признаки посещенности вершин – нулевые значения.

Начинаем с произвольной вершины v_0, присваиваем ей номер посещения, равный 1, и выбираем произвольное ребро (v_0, *w*). Ребро (v_0, *w*) помечается как «прямое», а вершина *w*, достигнутая из v_0, получает номер посещения, равный 2. После этого переходим в вершину *w*.

Пусть в результате выполнения нескольких шагов этого процесса мы пришли в вершину *x*, и пусть `Num` — последний присвоенный номер посещения. Возможны следующие ситуации:

– имеются непомеченные ребра, инцидентные этой вершине;
– все ребра, инцидентные вершине *x*, помечены.

В первом случае анализируем первое из непомеченных ребер (*x, y*). Если у вершины *y* уже есть номер посещения, то это ребро помечается как «обратное» и продолжается поиск непомеченного ребра, инцидентного вершине *x*. Если вершина *y* не имеет номера посещения, то переходим в эту вершину, увеличивая число ее посещений на единицу и помечая ребро (*x, y*) как «прямое». Вершина *y* считается получившей свой номер посещения из вершины *x*. На следующем шаге начинаем просматривать ребра, инцидентные вершине *y*.

Во втором случае возвращаемся в вершину, из которой *x* получила свой номер посещения. Процесс закончится, когда все ребра будут помечены и произойдет возвращение в вершину v_0.

Описанный процесс можно реализовать так, чтобы время работы соответствующего алгоритма составляло $O(M + N)$, где *M* — число ребер графа, *N* — число вершин графа. Для этого по мере обработки графа необходимо

формировать четыре списка: список посещений `NumVisit`, список имен вершин *F*, список ориентированных «прямых» ребер *T*, список ориентированных «обратных» ребер *B*. Ребра графа получают ориентацию в процессе работы алгоритма. Если ребро (*x, y*) помечается из вершины *x* как «прямое», то в *T* заносится дуга (*x, y*), а если как «обратное», то эта дуга заносится в список *B*. Элементом списка `NumVisit` является номер посещения вершины, а элементом списка *F* – имя вершины, из которой соответствующая вершина получила свой номер посещения.

5.3.1.2. Алгоритмы поиска в глубину в неориентированном связном графе

Алгоритм поиска через списки. Введем следующие параметры: `Num` — последний присвоенный номер посещения, *p* — указатель конца стека *Q*, т. е. *Q(p)* — вершина стека *Q*. В начале работы алгоритма зададим эти параметры: `Num=1; numVisit(v0)=Num; Q(p)=v0; F(v0)=0; T=0; B= 0; p= 1.` Алгоритм состоит из следующих шагов.

1. $v = Q(p)$.
2. Производится поиск вершины w, такой что ребро (*v*, *w*) является непомеченным; если такое ребро найдено, то перейти к шагу 4, иначе – к шагу 5.
3. Если вершина w имеет номер посещения, то пометить ребро (*v*, *w*) как «обратное» и занести в список *B*. Перейти к шагу 3 и продолжить просмотр списка смежных вершин. Иначе увеличить *Num* на единицу и добавить его значение в список *NumVisit*, добавить в список F значение *v*, пометить ребро (*v*, *w*) как «прямое» и занести в список *T*. Кроме того, увеличить на единицу значение указателя стека *p* и, используя обновленное значение указателя, занести в стек *Q* вершину *w*. После этого перейти к шагу 2.
4. Уменьшить на единицу значение указателя стека *p*, удалив тем самым вершину *v* из стека *Q*. Если $p = 0$, то завершить выполнение алгоритма, иначе перейти к шагу 2.

Алгоритм поиска пути между двумя вершинами. В алгоритме используется вспомогательный стековый массив вершин *Q, q* — число элементов в стеке, массив меток вершин *L(n)* и массивы меток ребер. Изначально все вершины считаются непомеченными. Алгоритм состоит из следующих шагов.

1. Поместить стартовую вершину в стек: $q = 1; Q(q) = s$.
2. Если стек не пуст, то берем вершину из стека $x = Q(q)$; $q = q - 1$, в противном случае пути нет и алгоритм заканчивает свою работу.
3. Если все смежные вершины просмотрены, то выполняем шаг 2. В противном случае рассматриваем очередную смеж-

ную с *x* вершину *y*. Если *y* — конечная вершина, то путь найден.

4. Проверяем наличие метки *y* вершины *y*. Если *L(y) > 0*, то помечаем ребро как «обратное», в противном случае ребро помечаем как «прямое» и присваиваем метку *L(y) = L(x) + 1*. Вершина y помещается в стек *q = q + 1; Q(q) = y*.
5. Выполняем шаг 3.

Алгоритм поиска по дереву. Еще один вариант алгоритма поиска в глубину основан на представлении ориентированного графа как дерева.

Выделим одну из вершин ориентированного графа. Построим для графа дерево универсального покрытия, в качестве корня которого будем использовать выделенную вершину. В качестве сыновей корневого узла рассматриваются вершины графа, находящиеся на концах инцидентных ему ребер. Затем этот процесс выполняется для каждого из сыновей корневого узла и т.д. Если в графе есть ориентированные циклы, то этот процесс будет бесконечным.

Будем предполагать, что ребра, выходящие из каждой вершины графа упорядочены (например, пронумерованы). Эта упорядоченность сохранится и для узлов дерева универсального покрытия. При обходе дерева сначала будет просматриваться корень, а затем – его поддеревья в порядке, определяемом нумерацией ведущих в них ребер. Такому обходу дерева соответствует обход графа. После удаления из маршрута этого обхода повторных посещений получим результат, соответствующий поиску в глубину.

Запишем этот алгоритм более подробно в терминах псевдоязыка (листинг 5.19).

Листинг 5.19. Алгоритм поиска в глубину

```
procedure FindDepth(n)
begin
  Node[n].Visit := True;  // отмечаем, что посетили
вершину
  Проверить, подходит ли вершина n
  Иначе begin
    for m ∈ Список соседних вершин do
      if Не посещали узел m then FindDepth(m)
  end;
end;
```

Изменение поля *Node[n].Visit* и анализ этого поля при выборе очередного узла обеспечивает нам прохождение каждого узла не более одного раза. Пусть N — число узлов. Тогда для каждого i-го узла нам придется просмотреть не более M_i дуг. Поэтому сложность алгоритма не более $N + \Sigma M_i = N + M$, т. е. $O(N + M)$.

В листинге 5.20 приведен полный текст рекурсивной процедуры поиска узла по имени *nameNode* в глубину.

Листинг 5.20. Рекурсивный поиск в глубину

```
int FindDepth(int n, string nameNode)
{
    int result = -1;
    VisitTrue(n);    // отметить посещенный
    if (Nodes[n].name == nameNode)
     result = n;
    else
    {
     int L = Nodes[n].Edge.Length;
     int i = -1; result = -1;
     while ((i < L - 1) && (result == -1))
     {
         int m = Nodes[n].Edge[++i].numNode;
         if (!Nodes[m].visit)
         {
           SetEdgeBlack(n, i); //закрасить дугу
           result = FindDepth(m, nameNode);
         }
     }
    }
    return result;
}

// Поиск в глубину
public int DepthSearch(int n, string nameNode)
```

```
{
   ClearVisit();
   int result = FindDepth(n, nameNode);
   return result;
}
```

Перед начальным вызовом рекурсивной процедуры неоходимо очистить поле *visit* у каждой вершины. Это делает функция *ClearVisit()* (листинг 5.21). Попутно эта функция также обнуляет поле номера визита *numVisit* и назначает всем дугам серый цвет.

Листинг 5.21. Первоначальная очистка узлов

```
void ClearVisit()
{
    int N = Nodes.Length; Lib.Num = 0;
    for (int i = 0; i < N; i++)
    {
     Nodes[i].visit = false;
     Nodes[i].numVisit = 0;
     Nodes[i].color = Color.White;
     int L = Nodes[i].Edge.Length;
     for (int j = 0; j < L; j++)
         Nodes[i].Edge[j].color = Color.Silver;
    }
}
```

При посещении узла мы должны изменить значение поля *visit* на *true*, и этим занимается функция *VisitTrue()* (листинг 5.22). Попутно эта функция назначает вершине очередной номер визита.

Листинг 5.22. Посещение узла

```
void VisitTrue(int n)  // отметить посещенный
{
    Nodes[n].visit = true;
    Lib.Num++;
```

```
        Nodes[n].numVisit = Lib.Num;
    }
```

Алгоритм поиска в глубину является одним из фундаментальных для графов, поэтому мы приведем и его нерекурсивную версию, которая использует стек. В терминах псевдоязыка алгоритм можно записать так, как приведено в листинг 5.23.

Листинг 5.23. Алгоритм нерекурсивного поиска в глубину

```
function DepthSearchStack(v, NameNode)
begin
  СТЕК := ∅;
  v ⇒ СТЕК;
  Node[v].Visit:=True; // посетили вершину
  while СТЕК <> ∅ do begin
    v ⇐ СТЕК;
    while (m ∈ Список соседних вершин) and
          Не найдена do
    если не посещали вершину m, то
    begin
      m ⇒ СТЕК;
      если Node[m].Name = NameNode то Найдено;
      Node[m].Visit:=True; // посетили вершину
    end;
  end;
end;
```

Двойной цикл обеспечивает сложность алгоритма не более $N + M$, т. е. $O(N + M)$. В листинге 5.24 приведен полный текст нерекурсивной процедуры поиска в глубину.

Листинг 5.24. Нерекурсивный поиск в глубину

```
public int DepthSearchStack(int n, string nameNode)
{
    Lib.myStack = new MyStack();
```

```
        ClearVisit();
        int result = -1;
        VisitTrue(n);    // отметить посещенный
        do
        {
         if (Nodes[n].name == nameNode)
             result = n;    // узел найден
         else
         {
             int i;
             // найти непосещенный узел
             int m = FindNotVisit(n, out i);
             if (m != -1)
             {
               SetEdgeBlack(n, i);//закрасить дугу
               Lib.myStack.Push(n);//поместить в стек
               VisitTrue(m);   // отметить посещенный
                n = m;
             }
             Else     //взять из стека
               n=(int)Lib.myStack.Pop();
         }
        }
        while (!(Lib.myStack.isEmpty() ||
             (result != -1)));
        return result;
    }
```

Листинг 5.25 содержит текст функции *FindNotVisit()*, предназначенной для поиска непосещенного узла.

Листинг 5.25. Поиск непосещенного узла

```
    int FindNotVisit(int t, out int i)
```

```
{
    // найти непосещенный узел
    int LL = Nodes[t].Edge.Length;
    bool Ok = false; i = -1; int result = -1;
    while ((i<LL-1) && !Ok)
      Ok = !Nodes[Nodes[t].Edge[++i].numNode].visit;
    if (Ok)
      result = Nodes[t].Edge[i].numNode;
    return result;
}
```

На рисунке 5.6 приведен результат поиска в глубину от вершины *Node0* до вершины *Node5*. Для каждого узла рядом с именем в скобках указан номер визита при поиске.

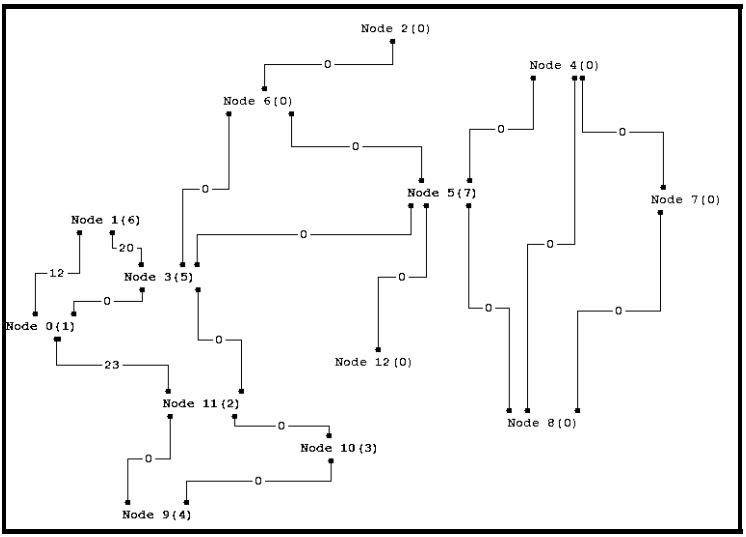


Рис. 5.6. Результат поиска в глубину

На рисунке 5.6 видно, что у узла *Node0* три соседа: *Node1*, *Node3* и *Node11*. Вначале алгоритм переходит в узел *Node1* и начинает просматривать соседей узла *Node1*. У этого узла только два соседа *Node0* и *Node3*. В *Node0* мы уже были, поэтому попадаем в узел *Node3*. У узла *Node3* пять соседей: *Node0*, *Node1*, *Node11*, *Node5*, *Node6*. Первый непосещенный узел *Node11*, но, побывав в соседних с ним узлах *Node9* и *Node10*, алгоритм

убеждается в том, что это не те узлы, и возвращается в список соседей узла `Node3`. Следующий соседний узел для `Node3` оказывается искомым. Алгоритм поиска в глубину хорош тем, что в момент нахождения искомого узла в стеке находятся номера узлов пути. Найденный путь не оптимален, т. е. не является кратчайшим. Достаточно легко улучшить этот результат, отсортировав предварительно для каждого узла массив дуг по возрастанию весов. На рисунке 5.7 представлен результат работы программы при поиске узла `Node7`.

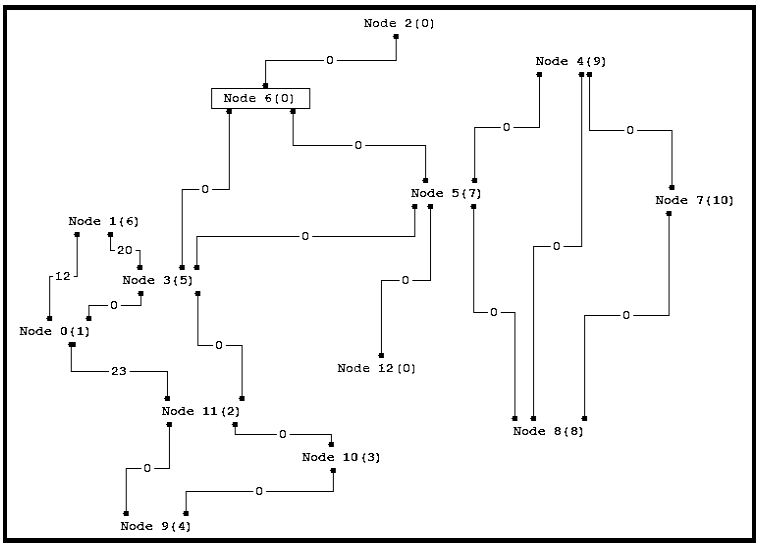


Рис. 5.7. Результат поиска узла `Node8`

При переходе от узла `Node1` к `Node2` алгоритм правильно выбрал первую дугу с минимальным весом 12. В результате переход от `Node1` к `Node4` дал минимальный суммарный вес 32. Но алгоритм подвел при переходе от `Node4` к `Node6`. Выбрав дугу к `Node7` с меньшим весом 23, алгоритм проиграл на суммарном весе, который мог бы быть 33, а стал 58.

5.3.2. Поиск в ширину

Другим фундаментальным алгоритмом поиска является *поиск в ширину* (*breadth first search*). Общая идея этого метода заключается в следующем: начиная поиск с какой-то вершины *n*, просматриваем список всех ее дуг. Если среди соседей находится искомый узел, то алгоритм завершает работу. Иначе переходим в первую соседнюю вершину и продолжаем алгоритм из этой вершины. Единственное отличие этого алгоритма от алгоритма поиска в глубину заключается в том, что мы вместо стека используем очередь, т. е.

при возврате при неудачном поиске по списку соседей возвращаемся в начало списка. Запишем этот алгоритм в терминах псевдокода более подробно (листинг 5.26).

Листинг 5.26. Алгоритм нерекурсивного поиска в ширину

```
function BreadthSearch(v, NameNode)
begin
  ОЧЕРЕДЬ := ∅;
  v ⇒ ОЧЕРЕДЬ;
  Node[v].Visit := True;  // посетили вершину
  while ОЧЕРЕДЬ <> ∅ do begin
    v ⇐ ОЧЕРЕДЬ;
    while (m ∈ Список соседних вершин) and Не найдена do
    если не посещали вершину m, то begin
      m ⇒ ОЧЕРЕДЬ;
      если Node[m].Name = NameNode то Найдено;
      Node[m].Visit:=True;  // посетили вершину
    end;
  end;
end;
```

Двойной цикл *while* по-прежнему обеспечивает сложность алгоритма $O(N + M)$. В листинге 5.27 приведен полный текст нерекурсивной процедуры поиска в ширину.

Листинг 5.27. Нерекурсивный поиск в ширину

```
public int BreadthSearch(int v, string nameNode)
// поиск в ширину
{
    ClearVisit();
    Lib.myStack = new MyStack();
    int result = -1;
```

```
        VisitTrue(v);     // отметить посещенный
        Lib.myStack.PushQueue(v);// поместить в очередь
        while (!Lib.myStack.isEmpty() && (result == -1))
        {
         v=(int)Lib.myStack.Pop();// взять из очереди
         int L = Nodes[v].Edge.Length;
         int i = -1;
         while ((i < L - 1) && (result == -1))
         {
             int m = Nodes[v].Edge[++i].numNode;
             if (!Nodes[m].visit)  // еще не посещали
             {
               SetEdgeBlack(v, i);// закрасить дугу
               // поместить в очередь
               Lib.myStack.PushQueue(m);
               if (Nodes[m].name == nameNode)
                   result = m;
               VisitTrue(m); // отметить посещенный
             }
         }
        }
        return result;
    }
```

На рисунке 5.8 приведен результат поиска в ширину от вершины *Node1* до вершины *Node6*. Для каждого узла рядом с именем в скобках указан номер визита при поиске.

Последовательность прохождения узлов при поиске в ширину, конечно же, не такая, как при поиске в глубину. Сначала просматриваем всех соседей узла *Node1*: *Node2*, *Node12*, *Node4*, *Node10*. Среди них искомого узла нет, поэтому смотрим непосещенных соседей узла *Node2*. Таких не оказалось. Нет непосещенных соседей и у узла *Node12*. Наконец, среди непосещенных соседей узла *Node4* находится искомый узел *Node6*.

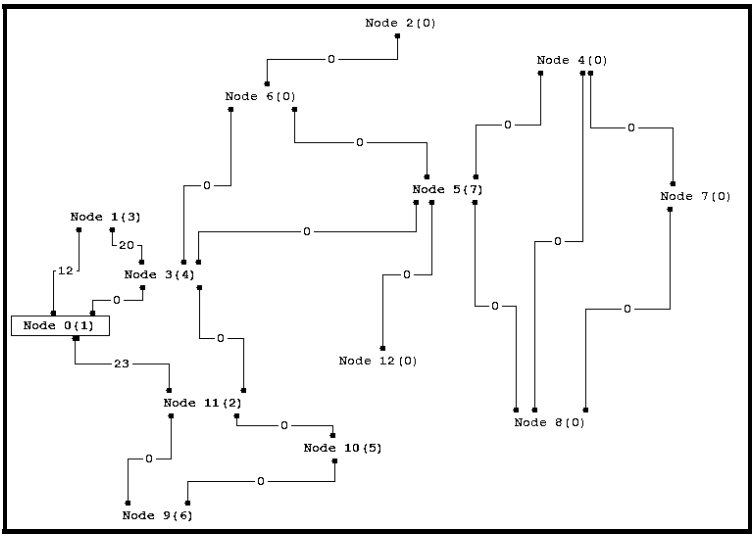


Рис. 5.8. Результат поиска в ширину

5.3.3. Остов графа

В любом связном неориентированном графе можно исключить часть ребер, превратив его в стягивающее дерево — *остов* (рис. 5.9). Достаточно легко доказать, что дерево с N вершинами имеет $(N-1)$ ребро. Мы будем строить стягивающее дерево с помощью слегка измененного алгоритма поиска в глубину.

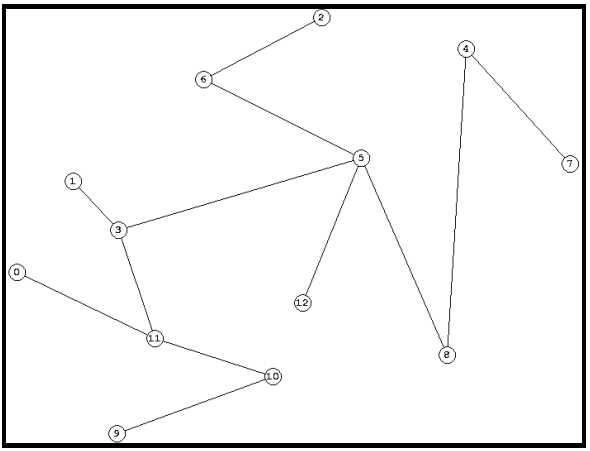


Рис. 5.9. Стягивающее дерево графа

В этом алгоритме нам потребуется какая-нибудь начальная вершина, но искать какую-нибудь вершину не надо. Поэтому вместо функции, используемой при поиске в глубину, напишем рекурсивную функцию *FindDepth* (листинг 5.28).

Листинг 5.28. Функция построения стягивающего дерева

```
void FindDepth(int n)
{
    VisitTrue(n); // отметить посещенный
    int LL = 0;
    if (Nodes[n].Edge != null)
    {
LL = Nodes[n].Edge.Length; int i = -1;
while (i < LL - 1)
{
    int m = Nodes[n].Edge[++i].numNode;
    if (!Nodes[m].visit)
    {
SetEdgeBlack(n, i); // закрасить ребро
FindDepth(m);
    }
}
    }
}

public void SetTreeDepth(int n)
// построение стягивающего дерева (остов) -
// поиск в глубину
{
    ClearVisit();
    FindDepth(n);
}
```

5.4. Кратчайшие пути

Задачи о кратчайших путях относятся к фундаментальным задачам комбинаторной оптимизации.

Пусть дан взвешенный неориентированный граф, и необходимо определить наилучший путь от узла *s* до узла *t*. Такая задача выглядит достаточно простой, но наилучший путь могут определять многие факторы, например:

- расстояние в километрах;
- время прохождения маршрута с учетом ограничений скорости;
- ожидаемая продолжительность поездки с учетом дорожных условий и плотности движения;
- задержки, вызванные проездом через города или объездом городов;
- число городов, которое необходимо посетить, например, в целях доставки грузов.

Известно, что все веса неотрицательны. Нужно найти наименьшую минимальную сумму весов $s \rightarrow i$ для всех $i = 1. . . n$ за время $O(N^2)$. На самом деле это ограничение можно ослабить требованием отсутствия циклов с отрицательным суммарным весом. В противном случае двигаясь от вершины *s* к вершине *t* и обходя такой цикл несколько раз, мы можем получить сколь угодно малую сумму весов.

Сформулируем две задачи о кратчайших путях.

1. Определить кратчайшие пути от вершины *s* до всех остальных вершин x_i.
2. Найти кратчайшие пути между всеми парами вершин.

При реализации алгоритмов предполагается, что матрица смежности, вообще говоря, не удовлетворяет условию треугольника $A_{ij} <= A_{ik} + A_{kj}$ и если между вершинами *i* и *j* отсутствует дуга, то вес дуги равен ∞ (при реализации – максимальное целое).

5.4.1. Волновой алгоритм

Волновой алгоритм поиска кратчайшего пути впервые был предложен Э.Ф. Муром в 1959 году. Относится к классу алгоритмов поиска в ширину и используется при компьютерной разводке соединительных проводников на поверхности электронных микросхем. Рассмотрим его для случая поиска кратчайшего пути между вершинами *s* и *t* невзвешенного графа.

Алгоритм использует два списка `OldFront` и `NewFront`, представляющих предыдущий и последующий «фронт волны», а также целую переменную *T* – текущее время. Отсчет времени ведется от нуля. Последовательность шагов алгоритма такова:

1) каждой вершине v_i приписывается целое число $T(v_i)$, которое будем называть волновой меткой; начальное значение волновой метки для всех вершин полагается равным –1;

2) в список `OldFront` заносится вершина *s*, а список `NewFront` первоначально считается пустым. Для вершины *s* устанавливается значение числовой метки, равное 0;
3) для каждой из вершин, входящих в `OldFront`, просматриваются инцидентные ей вершины u_j, и если они имеют волновые метки, равные -1, то они заменяются на $T + 1$ с добавлением соответствующих вершин в список `NewFront`;
4) если список `NewFront` пуст, то алгоритм завершается с результатом «нет решения»;
5) если одна из вершин списка `NewFront` совпадает с конечной вершиной *t*, то это означает, что найден кратчайший путь между *s* и *t* с $T + 1$ промежуточными ребрами, и алгоритм завершается с результатом «решение найдено»;
6) если алгоритм не завершен, то содержимое списка `NewFront` переносится в список `OldFront`, список `NewFront` снова полагается пустым, значение переменной *T* увеличивается на 1 и выполняется переход на шаг 4.

Примечание

Имеется различие в выполнении шага 3 для неоринтированных и ориентированных графов: для первых учитываются все инцидентные им вершины, а для вторых только вершины, в которые ведут дуги, исходящие из данной вершины.

На самом деле волновой алгоритм в общем случае определяет множество путей одинаковой и наименьшей длины между двумя заданными вершинами. Восстановить один из них можно следующим образом. Начинаем просмотр с вершины *t* и ищем среди ее соседей любую вершину с волновой меткой, на единицу меньшей, чем у вершины *t*. Среди соседей последней – вершину с меткой на два меньшей, чем у *t,* и далее вплоть до вершины *s*. Восстановление пути будет происходить быстрее, если на шаге 3 сохранять метку вершины, из которой «волна» пришла в данную вершину. Работа волнового алгоритма иллюстрируется рисунком 5.10.

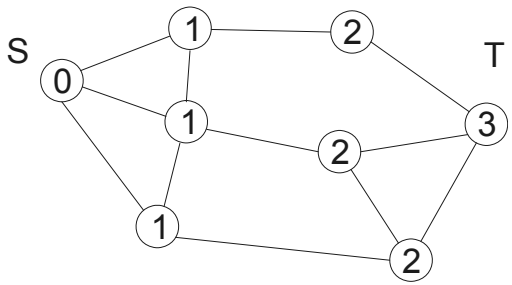


Рис. 5.10. Разметка графа после выполнения волнового алгоритма

5.4.2. Алгоритм Дейкстры

Наиболее эффективный алгоритм поиска кратчайших путей в графах с неотрицательными весами ребер дал Эдсгер В. Дейкстра [19]. Этот алгоритм основан на том, что каждому узлу приписывается расстояние `Dist` от исходного узла и признак возможности изменять это расстояние `Visit`, которые меняются в процессе работы алгоритма. Алгоритм основан на итерационном уточнении значений расстояний от исходных больших величин до конечных, равных длинам соответствующих кратчайших путей. Основная идея уточнения основана на простой формуле $Dist(x_i) = Min(Dist(x_i), Dist(p) + Ap_i)$, геометрический смысл которой представлен на рис. 5.11.

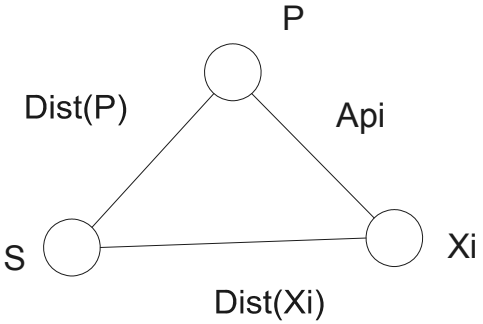


Рис. 5.11. Геометрическая интерпретация идеи уточнения расстояния

Если между вершинами p и x_i нет ребра, то $A_{pi} = \infty$ и $Dist(x_i)$ не меняется. Если же между этими вершинами есть ребро и $Dist(p)$ уже достигло минимального значения, то $Dist(x_i)$ также принимает минимальное значение.

В начале работы алгоритма выделенным является только узел s. На каждом шаге алгоритма в число выделенных добавляется один узел, поэтому множество выделенных узлов постепенно расширяется и, в конце концов, выделенными оказываются все узлы. В процессе выполнения алгоритма для каждого выделенного узла в массиве расстояний хранится наименьшая длина пути от исходного узла. При этом известно, что минимальное значение длины достигается на пути, проходящем только через выделенные узлы. Наконец, для каждого еще не выделенного узла в массиве расстояний хранится наименьшая длина пути, который проходит только через выделенные узлы.

Выбор очередного узла для включения в число выделенных производится следующим образом. Среди всех невыделенных узлов выбирается узел с минимальным значением расстояния от исходной вершины. Путь, соответствующий этому расстоянию, для этого узла является кратчайшим, так как проходит только через выделенные узлы, обладающие таким же свойством.

Отметим, что это утверждение справедливо только при условии неотрицательности весов ребер.

После добавления очередного выделенного узла необходимо произвести корректировку данных для невыделенных узлов. Достаточно при этом учесть лишь ребра, в которых новая вершина является последней, а это легко сделать, т. к. минимальная длина пути в новый узел уже известна.

Если для хранения множества выделенных узлов задан массив логического типа, то добавление одного узла к числу выделенных требует времени $O(N)$. В процессе работы алгоритма элементы массива расстояний переходят из состояния «можно изменять» в состояние «менять нельзя». Вначале только у элемента `Dist[s],` имеющего значение 0, ставится пометка «менять нельзя», и на каждом шаге итерационного процесса такая пометка ставится еще у одного элемента. Процесс заканчивается, когда у всех элементов будет стоять пометка «менять нельзя», т. е. за $(N - 1)$ шаг.

Для реализации этого алгоритма уточним структуру, описывающую узлы (листинг 5.30).

Листинг 5.30. Уточненная структура узлов графа

```
public class TNode
{
   public string name;  //
   public TEdge[] Edge; // ребра
   public bool visit;
   public int x, y;     //
   public int numVisit;
   public Color color;
   public int dist, oldDist;
}
```

В классе узла `TNode` появилось поле `dist`, предназначенное для хранения расстояния до узла *s*.

Приведем последовательность шагов для алгоритма Дейкстры при $A_{ij} \geq 0$:

1. Присвоение начальных значений. Положить `dist(s)` $= 0$ и считать этот узел помеченным `visit(s) = true`, т. е. в дальнейшем `dist(s)` не изменяется. Положить `dist`$(x_i) = \infty$. Положить $p = s$.

2. Обновление *dist*. Для всех непомеченных узлов $xi \in \Gamma(p)$ уточнить *dist* по формуле: $dist(x_i) = Min(dist(x_i), dist(p) + Ap_i)$.
3. Отметить один узел. Среди всех непомеченных узлов найти такой, для которого $dist(x_i^*) = Min(dist(x_i))$, и пометить его.
4. Положить $p = x_i^*$.
5. Если $p = s$, то путь найлен, иначе перейти к шагу 2.

В листинге 5.31 приведена нерекурсивная функция, реализующая этот алгоритм.

Листинг 5.31. Алгоритм Дейкстры

```
public int Dijkst(int s, int t)
{
   int result;
   ClearVisit(); SetMatr();
   int N = Nodes.Length;  // Инициализация
   for (int i = 0; i <= N - 1; i++)
      Nodes[i].dist = 0xFFFFF;
   Nodes[s].dist = 0; VisitTrue(s);
   int p = s;
   do
   {
      p = FindMinDist(p);
      VisitTrue(p);  // обновление и найти
   }
   while (p != t);
   result = Nodes[p].dist;
   PathToStack(s, p);
   return result;
}
```

Эта функция возвращает минимальное расстояние от узла *s* до узла *t*. Шаги 2 и 3 реализует функция *FindMinDist()* (листинг 5.32).

Листинг 5.32. Уточнение Dist и определение Min

```
int FindMinDist(int p)
{
    int MinDist = 0xFFFFF;
    int result = 0;
    int N = Nodes.Length;
    for (int i = 0; i <= N - 1; i++)
    {
if (!Nodes[i].visit)
{
    Nodes[i].dist = Math.Min(Nodes[i].dist,
Nodes[p].dist + A[p, i]);
    if (Nodes[i].dist < MinDist)
    {
MinDist = Nodes[i].dist; result = i;
    }
}
    }
    return result;
}
```

Алгоритм Дейкстры не определяет минимальный путь, т. е. последовательность вершин, по которым надо пройти от *s* до *t*. Но этот путь можно получить с помощью рекурсивного соотношения $Dist(x_i^*) + A(x_i^*, x_i) = Dist(x_i)$, т. к. вершина x_i^* предшествует вершине x_i на минимальном пути. Эту рекурсию реализует функция *PathToStack()*, которая помещает вершины минимального пути в стек (листинг 5.33).

Листинг 5.33. Определение минимального пути

```
void PathToStack(int s, int p)
{
```

```
        Lib.myStack = new MyStack();
        int N = Nodes.Length;
        while (p != s)
        {
    Lib.myStack.Push(p); // положить в стек
    int i = -1; bool Ok = false;
    while ((i < N - 1) && !Ok)
        Ok = (++i != p) && (Nodes[p].dist ==
    Nodes[i].dist + A[i, p]);
    p = i;
        }
    }
```

На рисунке 5.12 показан результат работы алгоритма при определении минимального пути от вершины *Node1* до *Node8*. В окне выведен стек с вершинами пути: 2, 4, 6, 9, 8.

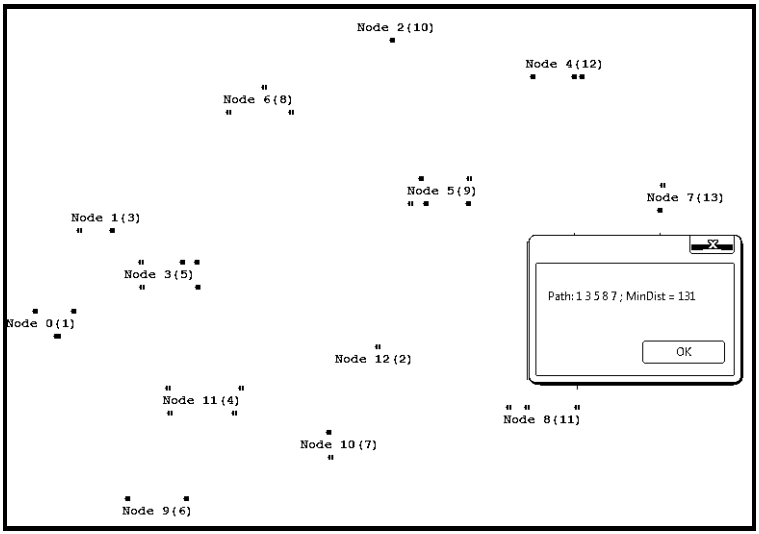


Рис. 5.12. Пример определения минимального пути от вершины *Node1* до *Node8*

Обратите внимание на то, что алгоритм не пошел из *Node4* в вершину *Node7* по пути длиной 23, более короткому, чем через *Node6*, и суммарно выиграл.

5.4.3. Алгоритм Форда-Мура-Беллмана

Алгоритм Дейкстры применим для матрицы смежности с неотрицательными коэффициентами ($A_{ij} >= 0$). Если же матрица *A* является матрицей стоимостей, то некоторые дуги могут иметь отрицательные веса. В этом случае достаточно эффективно работает алгоритм Форда-Мура-Беллмана.

Этот алгоритм ищет в бесконтурном графе кратчайшие пути от заданной вершины до всех остальных путем непосредственного решения уравнения Беллмана (уравнения динамического программирования). Алгоритм далек от того, чтобы быть оптимальным.

Рассмотрим пронумерованное множество из *N* городов, для каждой пары из которых в матрице смежности *A* хранится целое число, представляющее цену прямого билета для проезда между этими городами. Такая матрица в общем случае не является симметричной, то есть стоимости проезда в прямом и обратном направлениях могут различаться. В качестве минимальной стоимости проезда из города *i* в город *j* будем рассматривать наименьшую сумму цен билетов для всех проездов между всеми парами городов на этом пути. Эта стоимость не может превосходить значения элемента *A[i, j]* матрицы смежности и может оказаться существенно меньше. Ставится задача нахождения такой минимальной стоимости для заданной пары городов.

Примечание

Маршрут длиной, большей *N*, очевидно, содержит цикл. Это соображение позволяет ограничить поиск минимума маршрутами с длинами, не превосходящими *N*. Число таких маршрутов конечно и стоимость проезда по любому из них ограничена снизу нулем. Поэтому маршрут с наименьшей стоимостью существует.

Выберем в качестве начального город с номером 1 и найдем наименьшие стоимости проезда из него во все остальные города. Будем обозначать через *MinS(1, s, к)* минимальную стоимость проезда из города 1 в город *s* с менее, чем *k*, пересадками. Тогда выполняется такое соотношение:

```
MinS(1,s,k+1)=Min(MinS(1,s,k),MinS(1,i,k) + A[i,s]),
i = 1..n
```

Это соотношение является основой для алгоритма динамического программирования, называемым алгоритмом Форда-Беллмана. Этот алгоритм для узла с номером *from* состоит из следующих шагов.

1. Для всех узлов, кроме *from*, назначаем расстояние *Dist* = *MaxInt*, а для узла *from* расстояние назначаем равным нулю.

2. Для каждого узла сохраняем значение кратчайшего пути на предыдущем шаге.
3. Для всех узлов просматриваем все ребра и если найдется ребро, для которого выполняется условие: *A* + *Node[NumNode]*. *OldDist* < *Dist*, где *A* — стоимость проезда по ребру, то заменяем расстояние, и при помощи признака *Ok* := *true* отмечаем изменение.
4. Если на шаге 3 признак *Ok* = *true*, то повторяем шаг 2, иначе заканчиваем алгоритм.

Реализация этого алгоритма представлена в листинге 5.34.

Листинг 5.34. Алгоритм Форда-Мура-Беллмана

```
public void Bellmann(int from)
// было ли на этом шаге изменено значение кратчайше-
го
// пути хоть до одной вершины
{
    int N = Nodes.Length;
    bool Ok;
    for (int i = 0; i <= N - 1; i++)
        if (i == from)
            Nodes[i].dist = 0;
        else
            Nodes[i].dist = 0xFFFFF;
    do
    {// сохраняем значение кратчайшего пути на пре-
дыдущем шаге
       for (int i = 0; i <= N - 1; i++)
          Nodes[i].oldDist = Nodes[i].dist;
       Ok = false;
       for (int i = 0; i <= N - 1; i++)
       {
```

```
                int LL = 0;
                if (Nodes[i].Edge != null)
                {
                   LL = Nodes[i].Edge.Length;
                   if (LL > 0)
                   // просматриваем ребра, входящие в те-
кущую вершину
                      for (int j = 0; j <= LL - 1; j++)
                      {
                         Edge = Nodes[i].Edge[j];
                         if (Edge.A +
                           Nodes[Edge.numNode].oldDist <
                           Nodes[i].dist)
                         // если нашли путь короче
                         {
                            Nodes[i].dist = Edge.A +
                            Nodes[Edge.numNode].oldDist;
                            // исправляем Dist
                            Ok = true;   // и ставим флаг
                         }
                   }
                }
           }
           while (Ok); // если очередная итерация прошла
неуспешно
           }
```

Этот алгоритм найдет решение за время $O(N^3)$.

5.5. Циклы на графах

В пп. 5.1 было дано определение цикла как замкнутого маршрута без повторяющихся ребер. При отсутствии в нем повторяющихся вершин, кроме совпадающих первой и последней, цикл называется простым. Очевидно, что если в графе существует маршрут, ведущий из вершины v_0 в v_n, то существует и простая цепь между этими вершинами. Действительно, такую простую цепь можно построить, удалив из маршрута все циклы.

5.5.1. Эйлеровы циклы

Цикл, проходящий через все ребра графа, называется *эйлеровым циклом.*

Теорема Эйлера

Связный неориентированный граф содержит эйлеров цикл тогда и только тогда, когда число его вершин, имеющих нечетную степень, равно 0 или 2.

Эта теорема была доказана Эйлером в 1736 г. считается первой в теории графов. На рисунке 5.13 представлен эйлеров путь для неориентированного графа из девяти узлов.

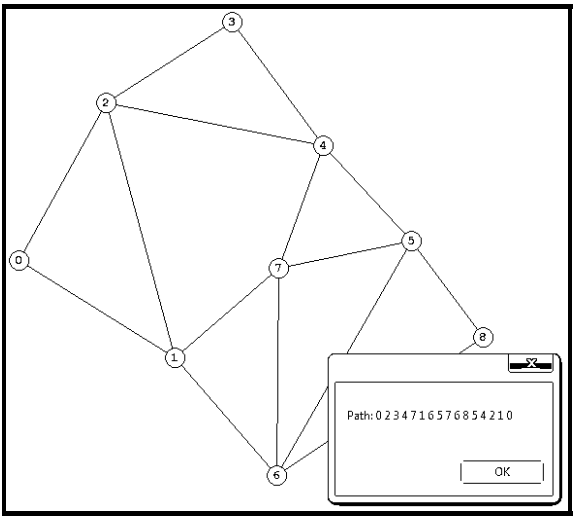


Рис. 5.13. Эйлеров путь

Рассмотрим алгоритм построения эйлерова пути. Для реализации алгоритма потребуется два стека: рабочий стек `myStack` и стек для эйлерова пути `path`. Алгоритм содержит следующие шаги.

1. Очистить признаки посещения узлов.
2. Выбрать в качестве исходного любой узел, например, узел с номером $v = 0$; поместить номер этого узла в стек *myStack*.
3. Если этот стек не пуст, то для текущего узла найти ребро, соединяющее его со смежным непосещенным узлом. Если стек пуст, то завершить выполнение алгоритма.
4. Если такое ребро найдено то, поместить в стек *myStack* номер соответствующего смежного узла, пометить ребро как пройденное и перейти на шаг 3. Если ребро не найдено, то выполнить шаг 5.
5. Извлечь номер узла из стека *myStack*, т. е. вернуться назад, поместить номер узла в стек пути *path* и перейти на шаг 3.

Результат работы алгоритма накапливается в стеке *path*. Два метода, совместно реализующих алгоритм построения эйлерова пути, представлены в листинге 5.35.

Листинг 5.35. Эйлеровы пути

```
int FindEdge(int v)
{
    int LL = 0;
    if (Nodes[v].Edge != null)
    {
       LL = Nodes[v].Edge.Length;
       bool Ok = false; int i = -1;
       while ((i < LL - 1) && !Ok)
         Ok = Nodes[v].Edge[++i].color !=
Color.Black;
           if (Ok) return i; else return -1;
    }
    else
        return -1;
}

public void PathEuler()
{
    ClearVisit();
```

```
        Lib.myStack = new MyStack();
        Lib.path = new MyStack();
        int v = 0;
        Lib.myStack.Push(v);        // положить в стек
        while (!Lib.myStack.isEmpty())
        {
            int i = FindEdge(v);
            if (i != -1)
            {
                int u = Nodes[v].Edge[i].numNode;
                Lib.myStack.Push(u);  // положить в стек
                SetEdgeBlack(v, i);   // закрасить ребра
                v = u;
            }
            else
            {
                v = (int)Lib.myStack.Pop();
                // взять из стека
                Lib.path.Push(v);    // положить в стек
            }
        }
    }
```

5.5.2. Гамильтонов цикл. Алгоритмы с возвратом

Напомним, что гамильтоновой цепью графа называется маршрут, который проходит через каждую вершину графа ровно один раз. Если такой маршрут замкнут, то он называется гамильтоновым циклом, а соответствующий граф – гамильтоновым графом. Такие названия цепей и циклов связаны с именем ирландского математика и физика-теоретика Уильяма Роуэна Гамильтона (Hamilton W.), который в 1859 г. предложил следующую игру-головоломку: требуется, переходя по очереди от одной вершины додекаэдра к другой вершине по его ребру, обойти все 20 вершин по одному разу и вернуться в начальную вершину.

Достаточные условия существования гамильтонова цикла были сформулированы в работах Оре (Ore O.) и Дирака (Dirac G. A.) в 1950-х годах. Но в наиболее общем виде они представлены в теореме, доказанной венгерским

математиком Лье Поша (Posa L.) в 1962 году. Отметим, что условие Поша существования гамильтонова цикла является достаточным, но не необходимым.

Теорема Поша

Пусть граф G имеет $p > 2$ вершин. Если для всякого n, $0 < n < (p - 1) / 2$, количество вершин со степенями, не превосходящими n, меньше n и для нечетного p количество вершин степени $(p - 1) / 2$ не превосходит $(p - 1) / 2$, то G — гамильтонов граф.

Следствие

В полном графе $G(V, E)$ всегда существует гамильтонов путь.

К сожалению, пока еще неизвестны эффективные методы решения поставленной задачи. Поэтому воспользуемся методом перебора с возвратом.

1. Начиная с определенной вершины (пускай это будет вершина с номером 1), будем продвигаться вперед по графу, включая очередное ребро в гамильтонов цикл.
2. При найденных первых k компонентах решения рассматриваем ребра, которые выходят из последней вершины. Если находим ребро, которое ведет в непосещенную ранее вершину, добавляем новую вершину в цикл — она становятся просмотренной. При этом $(k + 1)$-ая компонента решения получена.
3. При отсутствии такой вершины возвращаемся к предыдущей и ищем другие смежные с ней.

Цикл считается найденным, если просмотрены все вершины графа, и из последней вершины можно попасть в начальную. Такой цикл можно вывести на экран и продолжить поиск других циклов.

Рекурсивная реализация этого алгоритма представлена в листинге 5.36.

Листинг 5.36. Гамильтонов цикл

```
string SetPath()
{
    string result = "";
    for (int j = 1; j < Nodes.Length; j++)
        result += Convert.ToString(Lib.arr[j]) + "
";
    return result;
}

void Gamilt(int k,
```

```
  System.Windows.Forms.ListBox.ObjectCollection
Items)
{
    int y = Lib.arr[k - 1];
    int LL = 0;
    if (Nodes[y].Edge != null)
    {
       LL = Nodes[y].Edge.Length;
       for (int i = 0; i < LL; i++)
       {
           int u = Nodes[y].Edge[i].numNode;
           if (k == Nodes.Length + 1)
              Items.Add(SetPath());
              // вывести путь
           else
              if (!Nodes[u].visit)
              {
                 Lib.arr[k] = u;
                 Nodes[u].visit = true;
                 // отметить посещенный
                 Gamilt(k + 1, Items);
                 Nodes[u].visit = false;
                 // отметить посещенный
              }
       }
    }
 }

public void
Gamilton(System.Windows.Forms.ListBox.ObjectCollecti
on Items)
{
   ClearVisit();
```

```
        int N = Nodes.Length; int v0 = 0;
        Lib.arr[1] = v0;
        VisitTrue(v0);   // отметить посещенный
        Gamilt(2, Items);
    }
```

Необходимо отметить, что информация о проходимом пути накапливается в массиве *arr[]*. При неуспешном выходе из рекурсивной процедуры *Gamilt()* переустанавливается значение поля *visit = false*.

Как и любой алгоритм с возвратом, данный алгоритм имеет экспоненциальную скорость роста.

Пример. Рассмотрим граф, у которого одна вершина изолирована, а ребра, связывающие остальные $(N-1)$ вершину, образуют полный граф. Функция *Gamilt*() никогда не найдет гамильтонова цикла, но она исследует все $(N-2)!$ путей, начинающихся в выбранной начальной вершине, каждый из которых использует $(N-1)$ рекурсивных вызовов. Следовательно, общее число рекурсивных вызовов равно $(N-1)!$.

На рисунке 5.14 представлен список гамильтоновых циклов для графа, содержащего 7 узлов.

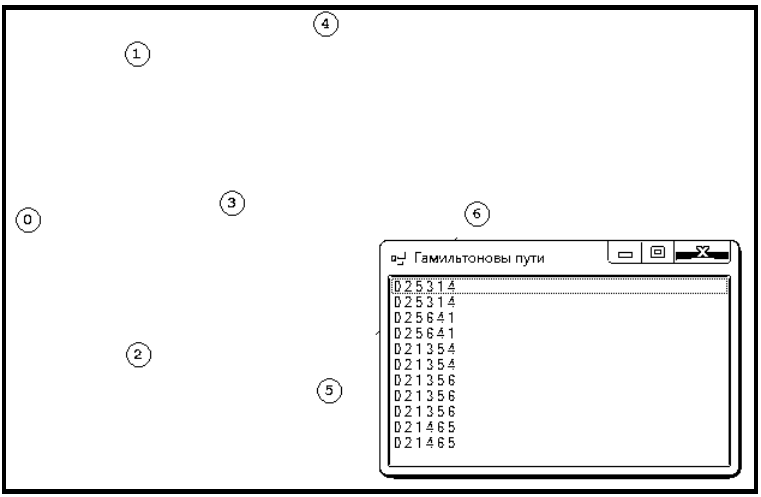


Рис. 5.14. Гамильтонов цикл

5.6. Гамильтоновы циклы и задача коммивояжера

Одной из формулировок задачи о поиске гамильтонова цикла является известная задача коммивояжера. Требуется найти кратчайший замкнутый

путь обхода нескольких городов, заданных своими координатами. Уже для относительно небольшого числа городов, порядка 60, эта задача оказывается практически неразрешимой путем простого перебора вариантов. Попытки ее решения дали толчок развитию новых вычислительных методов, в том числе нейронных сетей и генетических алгоритмов.

Каждый вариант решения данной задачи представляет собой строку, где на *j*-ом месте стоит номер *j*-го по порядку обхода города. Таким образом, в этой задаче *N* параметров, причем не все комбинации значений допустимы. Эта задача NP-полная (задача с нелинейной полиномиальной оценкой числа итераций). Генетический алгоритм используется для нахождения приблизительно оптимального пути за линейное время. В любой момент времени коммивояжер может быть только в каком-то городе. Города могут посещаться только однажды.

Проблема коммивояжера может быть успешно решена аналитическими методами для небольшого количества городов. Существенным недостатком этих методов является то, что при увеличении числа городов вычислительная сложность решения этой проблемы существенно возрастает, и при некоторых условиях она уже не может быть решена за линейное время или решается, но не оптимально.

Самые простые из аналитических методов решений – это полный перебор, кратчайший незамкнутый путь и метод ветвей и границ

Полный перебор крайне неэффективен с точки зрения вычислительной сложности. Количество всех просматриваемых решений равно $N!$, где N — количество городов. Преимущество этого метода заключается в том, что полученное с его помощью решение является однозначно оптимальным.

Метод кратчайшего незамкнутого пути заключается в следующем способе построения пути. Соединяются ребром две ближайшие точки, затем отыскивается точка, ближайшая к любой из уже рассмотренных точек, и соединяется с ней и т. д., до исчерпания всех точек. Главный недостаток этого метода состоит в том, что этот метод одиночной связи приводит к «серпантинным» или «цепным» решениям. Также имеет экспоненциальную скорость роста.

5.7. Комбинаторные задачи на графах

5.7.1. Минимальная раскраска графа

Задача о раскраске графа, как уже говорилось выше, связана с разбиением множества вершин графа на подмножества. Свойство вершин, положенное в основу такого разбиения, условно называют цветом. Соотвественно, *k*-раскраской графа *G* называется произвольная функция *f*, отображающая множество вершин графа *G* в некоторое множество *A* из *k* элементов:

$$f:\ V \to A = \{a_1, a_2, \ldots, a_k\}.$$

В качестве элементов множества A обычно используют отрезок натурального ряда $\{1, 2, \ldots, k\}$. Если $f(u) \neq f(v)$ для любых смежных вершин u и v графа, то раскраска называется правильной. Иначе говоря, концевые вершины любого ребра должны быть окрашены в разные цвета. Граф, для которого существует правильная k-раскраска, называется *k-раскрашиваемым.*

В графе на рисунке 5.15, состоящего из пяти узлов, для закраски использовалось пять красок с номерами от 1 до 5.

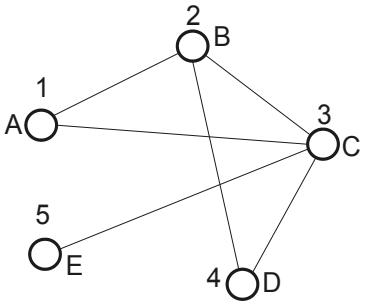


Рис. 5.15. 5-раскрашиваемый граф (раскраска правильная)

На рисунке 5.16 для правильной раскраски того же графа потребовалось три краски.

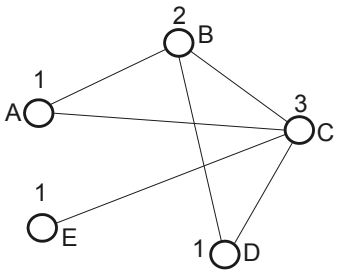


Рис. 5.16. 3-раскрашиваемый граф (правильная раскраска)

Минимальное число красок, обеспечивающее правильную раскраску графа, называется его *хроматическим числом,* а соответствующий граф называется *k-хроматическим* (используемое обозначение $\lambda(G) = k$).

Правильную k-раскраску графа G можно рассматривать как разбиение множества вершин графа G на не более чем k непустых множеств, которые называются *цветными классами*:

$$V = V_1 \, U \ldots V_k.$$

Каждый цветной класс является независимым множеством. Для полного графа хроматическое число равно числу его вершин. Очевидно, что для цикла с четным числом вершин хроматическое число равно двум, а для цикла с нечетным числом вершин – трем. Для пустого цикла это число полагают равным единице. Граф с хроматическим числом, равным двум, называется *бихроматическим*.

Теорема Кенига

Непустой граф является бихроматическим тогда и только тогда, когда он не содержит циклов нечетной длины.

Следствие 1.

Любое дерево бихроматично.

Следствие 2.

Любой двудольный граф бихроматичен.

5.7.2. Приближенные алгоритмы раскраски графа

Задачи о поиске хроматического числа является NP-сложной, т. е. не существует эффективных полиномиальных алгоритмов. Рассмотрим несколько приближенных алгоритмов.

5.7.2.1. Алгоритм последовательной раскраски вершин.

Это субоптимальный, приближенный алгоритм, который дает раскраску, близкую к минимуму.

1. Произвольной вершине графа G приписываем цвет 1.
2. Пусть раскрашены i вершин графа G в цвета от 1 до i, где $i \geq 1$. Произвольной неокрашенной вершине v_{i+1} приписываем минимальный цвет неиспользованной при раскраске смежных вершин.

Алгоритм последовательной раскраски зависит от способа перебора вершин.

На рисунке 5.17 приведен пример правильной раскраски, если последовательность такова: (A, B, G, D, C, F).

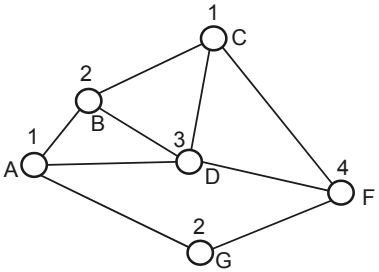


Рис. 5.17. Последовательность (A, B, G, D, C, F)

Этот же граф может быть раскрашен иначе (рис. 5.18).

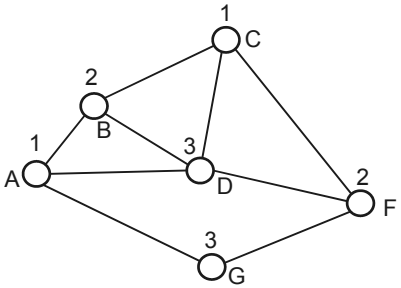


Рис. 5.18. Последовательность (A, B, D, C, F, G)

5.7.2.2. Алгоритм упорядочивания вершин

Этот алгоритм последовательной раскраски основан на методе упорядочивания вершин «наибольшие – первыми» или «наименьшие – первыми».

Метод «наибольшие – первыми» состоит в следующем. Упорядочиваем вершины графа G в порядке невозрастания их степеней. Если две вершины имеют одинаковые степени, то вычисляем двушаговые степени вершины v_i как число маршрутов длины 2, исходящих из этой вершины.

В методе «наименьшие – первыми» сначала выбираем в исходном графе вершину с наименьшей степенью и присваиваем ей номер p. Удаляем эту вершину со всеми инцидентными ей ребрами. В полученном графе находим вершину с наименьшей степенью и присваиваем ей номер $p - 1$ и т. д.

На рисунке 5.19 показан пример раскраски по методу «наибольшие – первыми».

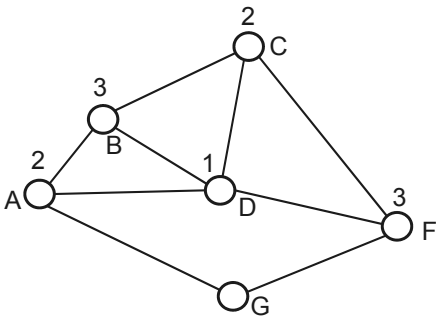


Рис. 5.19. Раскраска по методу «наибольшие – первыми»: (D, A, B, C, E, F)

На рисунке 5.20 показан пример раскраски того же графа по методу «наименьшие – первыми»

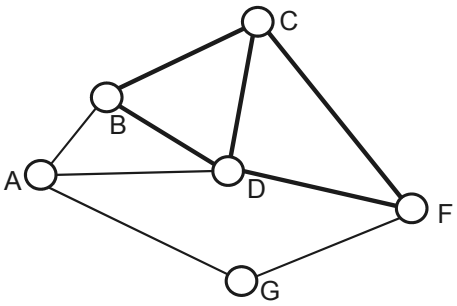


Рис. 5.20. Раскраска по методу «наименьшие – первыми»: (D, C, E, B, A, F)

Опишем алгоритм решения по шагам:

1. Вычислить степени вершин и отсортировать их в порядке невозрастания их степеней. Положить $K = 0$.
2. Просмотреть вершины в порядке невозрастания степеней и окрасить первую неокрашенную вершину в цвет номер K.
3. Просмотреть вершины в порядке невозрастания степеней и окрасить в цвет номер K все вершины, которые несмежны вершинам, уже окрашенным в цвет номер K.
4. Если все вершины окрашены, то K — искомое хроматическое число. Иначе $K = K + 1$ и переход к пункту 2.

Число использованных цветов будет приближенным значением хроматического числа графа.

Для реализации алгоритма требуется упорядочить вершины в порядке невозрастания степеней. Эта сортировка реализована для простоты пузырьковым способом, но при сортировке переставляются не сами вершины, а элементы вспомогательного массива V, в котором содержатся номера следования вершин (листинг 5.37).

Листинг 5.37. Сортировка вершин по невозрастанию степеней

```
void SortNumEdge(ref int[] V)
{
    int N = Nodes.Length;
    int L1, L2;
    for (int i = 0; i <= N - 1; i++) V[i] = i;
    for (int i = 0; i <= N - 2; i++)
        for (int j = N - 1; j >= i + 1; j--)
```

```
        {
            if (Nodes[V[j]].Edge != null)
                L1 = Nodes[V[j]].Edge.Length;
            else
                L1 = 0;
            if (Nodes[V[j - 1]].Edge != null)
                L2 = Nodes[V[j - 1]].Edge.Length;
            else
                L2 = 0;
            if (L1 < L2)
            {
                int t = V[j]; V[j] = V[j - 1];
                V[j - 1] = t;
            }
        }}
```

Перед реализацией самого алгоритма, представленного в листинге 5.38, задаются начальные значения динамического массива порядка следования вершин *V[i],* эти элементы сортируются процедурой *SortNumEdge(V),* процедурой *SetMatr()* задаются элементы матрицы смежности *A[i,j],* и задается начальное значение номера цвета $k = 0$.

Листинг 5.38. Функции раскраски графа и вычисления хроматического числа

```
public int SetColor()  // переборный алгоритм закраски
{
   int N = Nodes.Length;
   int[] V = new int[N];
   for (int i = 0; i <= N - 1; i++)
      V[i] = i;
   SortNumEdge(ref V);
```

```
   ClearVisit(); SetMatr();
   bool Ok;
   int k = 0;
   do
   {
      Ok = false; int i = -1;
      while ((i < N - 1) && !Ok)
      Ok = !Nodes[V[++i]].visit;
      if (Ok)
      {      // найдена неокрашенная
         VisitTrue(V[i]);          // окрасили
         Nodes[V[i]].color = Colors[k];
         for (i = 0; i <= N - 1; i++)
            if (!Nodes[V[i]].visit && IsBorder(V[i],
k))
            {     // не окрашена и нет соседей цвета
k
               VisitTrue(V[i]);  // окрашено
               Nodes[V[i]].color = Colors[k];
            }
         k++;
      }
   } while (Ok);
   return k;
}
```

В листинге 5.39 приведен текст функции, определяющей, есть ли у *i*-й вершины смежные соседи *k*-го цвета.

Листинг 5.39. Поиск несмежных вершин

```
bool IsBorder(int i, int k)
{
```

```
        int j = -1; bool result = false;
        int N = Nodes.Length;
        while ((j < N - 1) && !result)
           result = Nodes[++j].visit && (Nodes[j].color
==
                 Colors[k]) && (A[i, j] != 0xFFFFF);
        return result;
    }
```

На рисунке 5.21 показан пример раскраски графа с 13-ю вершинами. В данном случае хроматическое число оказалось равным 4.

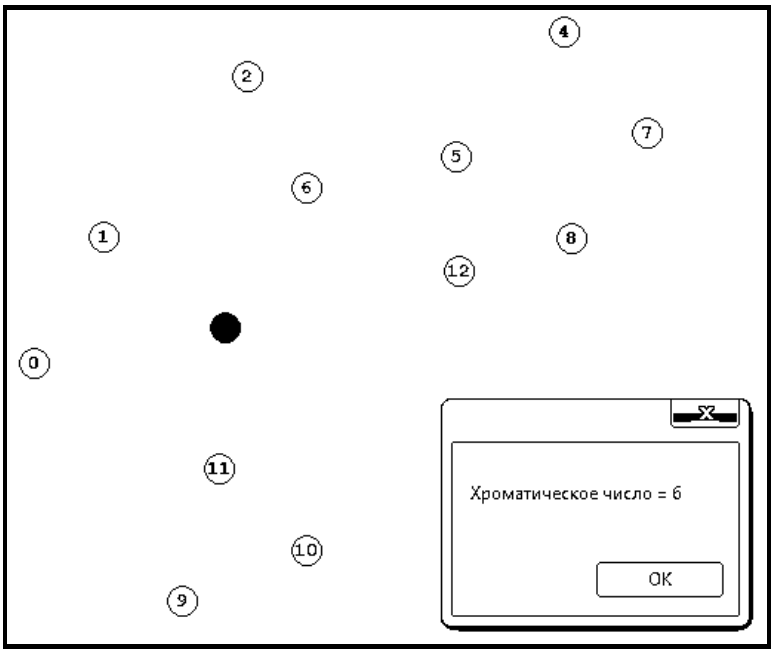


Рис. 5.21. Пример раскраски

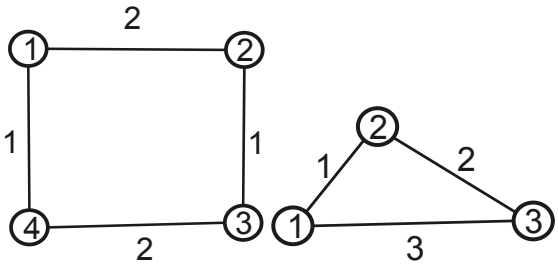
5.7.2.3. Раскраска ребер

Задачу раскраски можно сформулировать и для ребер графа. Реберной *k*-раскраской называется некоторая функция φ, задающая отображения множества ребер *E*:

$$\varphi: \quad E \to A = \{a_1, \ldots, a_k\}.$$

Реберная раскраска называется правильной, если все ребра, инцидентные любой вершине графа имеют разные цвета. Граф, для которого существует правильная реберная k-раскраска называется *k-раскрашиваемым*. Минимальное число k, при котором существует правильная реберная k-раскраска называется *реберным хроматическим числом* или *индексом* (используется обозначение $X(G) = k$). Граф G называется *реберно k-хроматическим*, если хроматический индекс равен k.

На рисунках 5.22 *а* и *б* приведены примеры графов, для раскраски ребер которых потребовалось 3 и 2 краски соответственно.



а *б*

Рис. 5.22. Графы, для раскраски ребер которых потребовалось 2 (а) и 3 (б) краски

Хроматический индекс для полного графа с четным числом вершин равен $X(K_{2n}) = 2n - 1$, а с нечетным числом вершин $X(K_{2n+1}) = 2n + 1$. Так, например, для полного графа с четырьмя вершинами (рис. 5.23) необходимо 3 краски для раскрашивания 6 ребер.

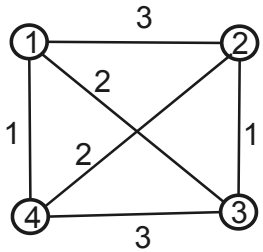


Рис. 5.23. Раскраска графа с четырьмя вершинами

5.8. Алгоритмы о связности графа

Понятие связности графа было введено в п. 5.1 настоящей главы. Напомним, что в связном графе для любой пары вершин существует соединяю-

щая их простая цепь. Далее будут рассмотрены некоторые алгоритмы, в которых существенным образом используется это понятие.

5.8.1. Топологическая сортировка

Дан ориентированный граф без циклов, т. е. имеется N узлов, пронумерованных от 0 до $N-1$, и из каждого узла выходят несколько ребер в другие узлы. Задача *топологической сортировки* формулируется следующим образом. Требуется упорядочить вершины графа так, чтобы конец любого ребра предшествовал его началу.

Такое упорядочение вершин всегда возможно. Действительно, из условия отсутствия циклов вытекает, что есть узел, из которого не выходят ребра. Выберем его в качестве исходного. Удаляя все ребра, ведущие в него, сводим задачу к графу с меньшим числом узлов и продолжаем рассуждение по индукции.

Напомним, что узлы ориентированного графа без циклов хранятся в экземплярах класса `TNode` (листинг 5.40).

Листинг 5.40. Структура узлов ориентированного графа

```
public class TNode
{
   public string name;  //
   public TEdge[] Edge; // ребра
   public bool visit;
   public int x, y;
   public int numVisit;
   public Color color;
   public int dist, oldDist;
}
```

В каждом `Node[i]` хранится динамический массив ребер `Edge`, элементы которого указывают на другие узлы. В поле `visit` мы будем, как обычно, хранить сведения о том, какие вершины пройдены, и корректировать его одновременно с прохождением узла.

Построим рекурсивный алгоритм, выполняющий топологическую сортировку не более чем за $O(N+M)$ действий, где M — число ребер графа. Программа будет помещать номера узлов в стек (листинг 5.41). Будем говорить, что последовательность вершин *корректна*, если:

– ни одна из вершин не посещалась дважды;
– для любой вершины, входящей в эту последовательность, все вершины, в которые ведут исходящие из нее дуги, были помещены в стек до нее.

Листинг 5.41. Топологическая сортировка

```
void SetVisit(int i)
{
    if (!Nodes[i].visit)
    {
        if (Nodes[i].Edge != null)
          for (int j=0;j<=Nodes[i].Edge.Length-
1;j++)
          {
              SetEdgeBlack(i, j); // закрасить ребро
              SetVisit(Nodes[i].Edge[j].numNode);
          }
        Lib.myStack.Push(i);
        VisitTrue(i);           // отметить песеще-
ние
    }
}
```

Основная программа вызывает процедуру *SetVisit()* для всех *N* узлов:

```
public void TopSort()
{
   int N = Nodes.Length;
   ClearVisit();
   Lib.myStack = new MyStack();
   for (int i = 0; i <= N - 1; i++)
      SetVisit(i);
}
```

Чтобы доказать конечность последовательности рекурсивных вызовов для какой-либо вершины, рассмотрим максимальную длину пути по дугам, выходящим из этой вершины. Будем называть эту длину *глубиной вершины*. Условие отсутствия циклов гарантирует, что эта величина конечна. Очевидно, что из вершины нулевой глубины дуг не выходит. Глубина конца вершины у конца дуги, по крайней мере, на единицу меньше, чем глубина вершины у ее начала. При работе процедуры `SetVisit(i)` все рекурсивные вызовы `SetVisit(j)` относятся к вершинам меньшей глубины.

На рисунке 5.24 представлены результаты работы.

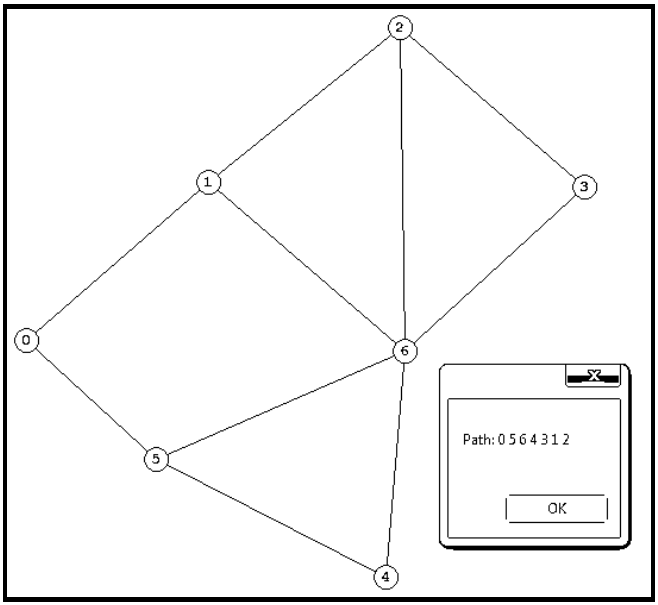


Рис. 5.24. Топологическая сортировка

5.8.2. Минимальное остовное дерево

Остовным деревом связного неориентированного графа называется его ациклический, т.е. не содержащий циклов, подграф, в который входят все его вершины. Остовное дерево может быть получено удалением из графа избыточных ребер с сохранением его связности. Если ребра графа снабжены весами, то можно ставить задачу построения *минимального остовного дерева,* то есть остовного дерева с минимально возможной суммой весов ребер.

В качестве практического примера решения этой задачи можно проблему прокладки линий коммуникаций между заданным числом объектов при следующих дополнительных условиях:

- известны «расстояния» между каждой парой объектов (это может быть геометрическое расстояние или стоимость прокладки коммуникаций между ними);
- объекты могут быть связаны как непосредственно, так и с участием произвольного количества промежуточных объектов;
- разветвления линий могут иметь место только в местах расположения объектов.

В такой постановке задача сводится к нахождению минимального остовного дерева во взвешенном графе, вершины которого соответствуют заданным объектам, а веса ребер равны «расстояниям» между ними. Возможность введения дополнительных объектов и, следовательно, дополнительных точек ветвления, позволяет минимизировать сумму «расстояний» между узлами коммуникационной сети. Если к тому же дополнительные объекты могут выбираться только из некоторого имеющегося множества, то получаем задачу построения минимального остовного дерева, кратчайшее дерево, покрывающего заданное подмножество вершин взвешенного графа. Задача в такой постановке известна как задача Штейнера, она является чрезвычайно сложной с вычислительной точки зрения и практически может быть решена лишь при небольшом числе дополнительных вершин. Существует, однако, достаточно эффективный приближенный алгоритм, строящий остовное дерево графа, длина которого превышает длину минимального дерева не более чем в два раза.

Задача поиска минимального остовного дерева для всего графа, в отличие от задачи Штейнера, допускает эффективное решение. Далее будут рассмотрены два алгоритма решения этой задачи.

1. Упорядочить ребра графа по возрастанию весов.
2. Выбрать ребро с минимальным весом, не образующее цикл с выбранными ранее ребрами. Занести выбранное ребро в список ребер строящегося остова.
3. Проверить, все ли вершины графа вошли в построенный остов. Если нет, то выполнить шаг 2.

Данный алгоритм имеет скорость роста *O(M),* где *M* — число ребер. Предположим, что ориентированный граф без циклов имеет узлы `Node[i]` с номерами *1...N*. Для каждого узла *i* известен динамический массив `edge[j]` выходящих из него ребер, каждый элемент которого указывает на номер узла, в который ведет одно из этих ребер. Напечатать все вершины в таком порядке, чтобы конец любого ребра был напечатан перед его началом.

Добавим к графу вершину 0, из которой ребра ведут в вершины *1, ... , N*. Если ее удастся напечатать с соблюдением правил, то тем самым все вершины будут напечатаны.

Алгоритм хранит путь, выходящий из нулевой вершины и идущий по ребрам графа. Переменная *L* отводится для длины этого пути. Путь образован

вершинами *vert[1], . . . , vert[L]* и ребрами, имеющими номера *edge[1], . . . , edge[L]*. Номер *edge[s]* относится к нумерации ребер, выходящих из вершины *vert[s]*. Тем самым для всех s должны выполняться неравенство:

```
edge[s] <= num[vert[s]]
```

и равенство:

```
vert[s+1] = dest [vert[s]] [edge[s]]
```

Впрочем, для последнего ребра сделаем исключение, разрешив ему указывать «в пустоту», т. е. *edge[L-1]* может равняться *num[vert[L-1]] + 1*.

В процессе работы алгоритм будет печатать номера вершин, соблюдая при этом требование: вершина печатается только после тех вершин, в которые из нее ведут ребра. Наконец, будет выполняться такое требование: вершины пути, кроме последней (т. е. *vert[0]..vert[L-1]*) не напечатаны, но свернув с пути налево, мы немедленно упираемся в напечатанную вершину.

Листинг 5.42. Алгоритм построения минимального остовного дерева

```
int L = 1;  vert[0] := 0;  edge[0] := 1;
while (! ((L = 1) && (edge[0] = n + 1)))
  if (edge[L-1]==num[vert[L-1]]+1)
  {
    // путь кончается, поэтому все вершины,
    // следующие за vert[L],
    // напечатаны - можно печатать vert[L-1]
    Console.WriteLine(vert[L-1]);
    edge[--L]++;
  }
  else
  {
    // edge[L-1] <= num[vert[L-1]],
    // путь кончается в вершине
    lastvert = dest[vert[L-1]][edge[L-1]];
    // последняя
    if (lastvert == напечатана) edge[L-1]++;
```

```
    else
    {
      vert[++L] = lastvert;  edge[L-1] = 1;
    }
  }
  // путь сразу же ведет в пустоту, поэтому все вер-
шины
  // левее, т. е. 1...n, напечатаны
```

Докажем, что если в графе нет циклов, то алгоритм заканчивает работу. Предположим, что это не так. Каждая вершина может печататься только один раз, поэтому с некоторого момента вершины не печатаются. В графе без циклов длина пути ограничена (одна и та же вершина не может входить дважды), поэтому, подождав еще, мы можем прийти к тому, что путь перестанет удлиняться. После этого может разве что увеличиваться *edge[L],* но и это не беспредельно.

5.8.3. Построение минимального остовного дерева

Алгоритм предназначен для нахождения минимального остовного дерева, т. е. такого подграфа, который бы имел столько же компонент связности, сколько и исходный, но не содержал бы петель и сумма весов всех его ребер была бы минимальной.

Вначале опишем алгоритм (возможно недостаточно строго), а затем обсудим, какой способ задания графа был бы наилучшим в данном случае, а также покажем, как от тех способов задания, которые мы использовали ранее, перейти к способу, применимому здесь.

Алгоритм Краскала выглядит так.

1. Отсортировать ребра графа по возрастанию весов.
2. Полагаем, что каждая вершина относится к своей компоненте связности.
3. Пройти ребра в «отсортированном» порядке. Для каждого ребра выполнить следующие действия:
   - если вершины, соединяемые данным ребром, лежат в разных компонентах связности, то объединить эти компоненты в одну, а рассматриваемое ребро добавить к минимальному остовному дереву;
   - если вершины, соединяемые данным ребром, лежат в одной компоненте связности, то исключить ребро из рассмотрения.

4. Если есть еще нерассмотренные ребра и не все компоненты связности объединены в одну, то перейти к шагу 3, иначе выйти из алгоритма.

В данном случае работать с матрицей весов неудобно, это выявляется уже на этапе упорядочивания ребер по весу, поэтому создадим временный массив ребер *Edges[]* с соответствующими весами. Каждый элемент этого массива имеет три поля: начальная вершина, конечная вершина и вес:

```
struct TEdges
// временная структура для ребра
{
   public int n1;      // № первой вершины
   public int n2;      // № второй вершины
   public int A;       // вес ребра
}
```

Упорядочиваем ребра по неубыванию весов пузырьковой сортировкой. Далее в алгоритме вводится массив *int[] Link*, элементы которого характеризуют номер компоненты связности соответствующих вершин (две вершины *u1*, *u2* лежат в одной компоненте связности, если и только если *Link[u1]* = *Link[u2]*).

В алгоритме (листинг 5.43) используется переменная *q*, которая инициализируется значением *N-1*, и затем при объединении двух компонент связности на шаге 3 *q* уменьшается на единицу, таким образом, условие *q* = 0 будет означать, что все вершины лежат в одной компоненте связности и работа алгоритма завершена. Найденное дерево будет определено с помощью стека, в который мы помещаем номера ребер.

Листинг 5.43. Алгоритм Краскала

```
public void Craskal()
{
    ClearVisit();
    Lib.myStack = new MyStack();
    int N = Nodes.Length; int m = 0;
    // формируем массив всех ребер
    for (int i = 0; i <= N - 1; i++)
    {
```

```
int L = 0;
if (Nodes[i].Edge != null)
{
    L = Nodes[i].Edge.Length;
    for (int j = 0; j <= L - 1; j++)
       if (!Find(i, Nodes[i].Edge[j].numNode))
       {
   Array.Resize<TEdges>(ref Edges, ++m);
   Edges[m - 1].n1 = i;
   Edges[m - 1].n2 =
     Nodes[i].Edge[j].numNode;
   Edges[m - 1].A = Nodes[i].Edge[j].A;
       }
}
    }
    TEdges h = new TEdges();
    // Сортируем ребра графа по возрастанию весов
    for (int i = 0; i <= m - 2; i++)
for (int j = m - 1; j >= i + 1; j--)
    if (Edges[j].A < Edges[j - 1].A)
    {
h = Edges[j]; Edges[j] = Edges[j - 1];
Edges[j - 1] = h;
    }
    int[] Link = new int[N];
   // Вначале все ребра в разных компонентах связно-
сти
    for (int i = 0; i <= N - 1; i++) Link[i] = i;
    int numEdge = 0; // номер ребра
    int q = N - 1;       // номер вершины
    while ((numEdge <= m - 1) && (q != 0))
```

```
    {
// если вершины в разных компонентах связности
if (Link[Edges[numEdge].n1] !=
    Link[Edges[numEdge].n2])
{
    int t = Edges[numEdge].n2;
    Lib.myStack.Push(numEdge);
    // поместить в стек номер ребра
    for (int j = 0; j <= N - 1; j++)
if (Link[j] == t)
    Link[j] = Link[Edges[numEdge].n1];
    // в один компонент связности
    q--;
}
numEdge++;
    }
    SetColorEdge(); // закраска ребер
}
```

Функция *SetColorEdge()* извлекает номера ребер из стека и перекрашивает ребра структуры *Node[i]* (листинг 5.44).

Листинг 5.44. Ракраска ребер минимального остова

```
void SetColorEdge()
{
    while (!Lib.myStack.isEmpty())
    {
        int t = (int)Lib.myStack.Pop();
        // взять из стека № ребра
        int N = Nodes.Length;
        for (int i = 0; i <= N - 1; i++)
        {
```

```
            int LL = 0;
            if (Nodes[i].Edge != null)
            {
              LL = Nodes[i].Edge.Length;
              for (int j = 0; j <= LL - 1; j++)
                if (((i == Edges[t].n1) &&
                  (Nodes[i].Edge[j].numNode ==
Edges[t].n2)) |
                  ((i == Edges[t].n2) &&
                   (Nodes[i].Edge[j].numNode ==
Edges[t].n1)))
                  SetEdgeBlack(i, j);
                  // закрасить ребро
             }
          }
      }
}
```

Результат работы алгоритма представлен на рисунке 5.25.

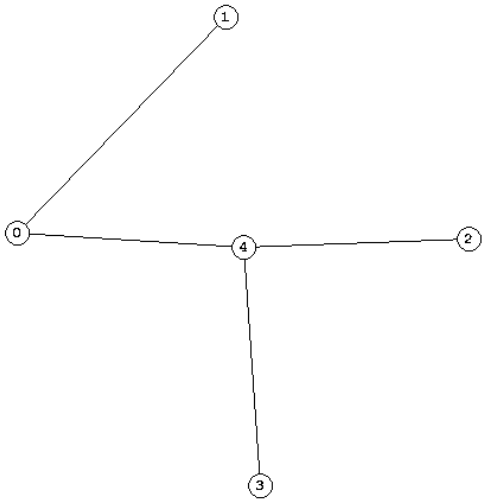


Рис. 5.25. Раскраска ребер минимального остова

5.8.4. Выделение компонент связности

Неориентированный граф – это набор точек (вершин), некоторые из которых соединены ребрами. Неориентированный граф можно считать частным случаем ориентированного графа, в котором для каждого ребра есть обратное ребро.

Связной компонентой вершины i называется множество всех тех вершин, в которые можно попасть из *i*, идя по ребрам графа. Поскольку граф неориентированный, отношение «*j* принадлежит связной компоненте *i*» является отношением эквивалентности.

Пусть дан неориентированный граф, для каждой вершины указано число соседей и массив номеров соседей, как в предыдущей задаче. Составить алгоритм, который по заданному *i* печатает все вершины связной компоненты *i* по одному разу (и только их). Число действий не должно превосходить *O*(общее число вершин и ребер в связной компоненте).

Программа (листинг 5.45) в процессе работы будет закрашивать некоторые вершины графа. Незакрашенной частью графа будем называть то, что останется, если выбросить все закрашенные вершины и ведущие в них ребра. Функция *SetEdge(v)* закрашивает связную компоненту *v* в незакрашенном графе и ничего не делает, если вершина *v* уже закрашена.

Листинг 5.45. Выделение связных вершин графа

```
void SetEdgeR(int v)
{
    if (!Nodes[v].visit)
    {
VisitTrue(v);
Lib.myStack.Push(v);
int LL = 0;
if (Nodes[v].Edge != null)
{
    LL = Nodes[v].Edge.Length;
    for (int i = 0; i <= LL - 1; i++)
SetEdgeR(Nodes[v].Edge[i].numNode);
}
    }
```

```
}
public void SetEdge(int s)
{
    ClearVisit();
    Lib.myStack = new MyStack();
    SetEdgeR(s);
}
```

Докажем, что эта функция работает правильно. В самом деле, ничего, кроме связной компоненты незакрашенного графа, она закрасить не может. Проверим, что вся она будет закрашена.

Пусть *k* — вершина, доступная из вершины *i* по пути *i*—*j*— ... —*k*, проходящему только по незакрашенным вершинам. Будем рассматривать только пути, не возвращающиеся снова в *i*. Из всех таких путей выберем путь с наименьшим *j* в порядке просмотра соседей в цикле процедуры. Тогда при рассмотрении предыдущих соседей ни одна из вершин *j*— ... —*k* не будет закрашена (иначе *j* не было бы минимальным), и потому *k* окажется в связной компоненте незакрашенного графа к моменту вызова *SetColor(j),* что и требовалось.

Чтобы установить конечность глубины рекурсии, заметим, что на каждом уровне рекурсии число незакрашенных вершин уменьшается хотя бы на 1. Оценим число действий. Каждая вершина закрашивается не более одного раза — при первом вызове *SetColor(i)* с данным *i*. Все последующие вызовы происходят при закрашивании соседей — количество таких вызовов не больше числа соседей и сводятся к проверке того, что вершина *i* уже закрашена. Первый же вызов состоит в просмотре всех соседей и рекурсивных вызовах *SetColor(j)* для них. Таким образом, общее число действий, связанных с вершиной *i*, не превосходит константы, умноженной на число ее соседей. Отсюда и вытекает требуемая оценка.

Выводы

Графы – это одна из самых важных структур данных, используемых в информатике для моделирования различных систем. Основным отношением между элементами графа – его узлами – является отношение смежности, определяющее свойство связности графа. Два узла графа являются смежными, если имеется соединяющее их ребро. Ребро, имеющее направленность, называется дугой, а граф, все ребра которого являются дугами, называется орграфом.

Деревья являются частным случаем графа. Так, остовное дерево получается из графа удалением максимального числа ребер с сохранением свойства связности. Также, как и в случае деревьев, существуют различные алгоритмы обхода узлов графа. Наиболее известными из них являются алгоритмы обхода графа в глубину и в ширину. Специальными видами обхода графа являются циклы Эйлера и Гамильтона.

Ребра графа могут быть снабжены дополнительными характеристиками – весами. Для взвешенного графа можно поставить целый ряд интересных задач, в частности задачи поиска минимального пути между двумя вершинами, поиска кратчайшего замкнутого пути, построения остовного дерева наименьшей стоимости. Для более изучения алгоритмов работы с графами можно рекомендовать монографии [26, 35, 45].

Упражнения

1. Пусть $G = (V, E)$ — связный неориентированный граф. Разработайте алгоритм для вычисления за время $O(V + E)$ пути в G, который проходит по каждому ребру из E по одному разу в каждом направлении.
2. Ориентированный граф $G = (V, E)$ обладает свойством односвязности (*singly connected*), если имеется не более одного пути для каждой пары вершин этого графа. Разработайте эффективный алгоритм для определения, является ли ориентированный граф односвязным.
3. Используя класс `Timing` (глава 1), сравните скорость поиска в графе с использованием алгоритмов обхода в ширину и в глубину.
4. Модифицируйте алгоритм построения эйлерова цикла так, чтобы не требовалась предварительная проверка четности степеней. Существование вершины нечетной степени должно обнаруживаться по ходу работы алгоритма.
5. Докажите, что при любом $N \geq 2$, где N – число вершин, в графе существует гамильтонов цикл.

Глава 6. Некоторые численные методы

6.1. Решение системы линейных уравнений методом Гаусса

Во многих прикладных задачах возникает необходимость решения систем линейных алгебраических уравнений вида:

$$Ax = f, \tag{6.1}$$

где A – матрица $m \times m$, $x = (x_1, x_2, \ldots, x_m)^T$ – искомый вектор, $f = (f_1, f_2, \ldots, f_m)^T$ – заданный вектор. Предполагается, что определитель матрицы A отличен от нуля, так что решение x существует и единственно. Для большинства вычислительных задач характерным является большой порядок матрицы A.

Методы численного решения системы линейных алгебраических уравнений делятся на две группы: прямые методы и итерационные методы. Итерационные методы состоят в том, что решение x системы находится как предел при $n \to \infty$ последовательных приближений $x^{(n)}$, где n – номер итерации. Обычно задается малое число $\varepsilon > 0$ и вычисления проводятся до тех пор, пока не будет выполнена оценка

$$x^{(n)} - x < \varepsilon. \tag{6.2}$$

Число итераций n является функцией точности $n = n(\varepsilon)$. Многие итерационные методы предназначены для матриц специального вида (трехдиагональные, симметричные, ленточные, большие разреженные и т. д.).

Наиболее известными прямыми способами решения системы (6.1) являются метод Крамера, основанный на вычислении определителей, и метод Гаусса, основанный на последовательном исключении неизвестных. При больших m первый способ требует порядка $m!$ арифметических действий, а второй – только $O(m^3)$ действий. Поэтому метод Гаусса широко используется при численном решении задач линейной алгебры. Прямые методы не предполагают, что матрица A имеет какой-либо специальный вид и ее порядок не превышает 100.

Запишем систему (6.1) в развернутом виде

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots a_{1m}x_m = f_1, \\ &a_{21}x_1 + a_{22}x_2 + \ldots a_{2m}x_m = f_2, \\ &\ldots a_{m1}x_1 + a_{m2}x_2 + \ldots a_{mm}x_m = f_m. \end{aligned} \tag{6.3}$$

Метод Гаусса решения системы (6.3) состоит в последовательном исключении неизвестных $x_1, x_2, \ldots, x_m$ из этой системы. Предположим, что $a_{11} \neq 0$. Поделив первое уравнение на a_{11}, получим

$$x_1 + c_{12}x_2 + \ldots + c_{1m}x_m = y_1, \qquad (6.4)$$

где

$$c_{1j} = \frac{a_{1j}}{a_{11}},\ j = 2,\ldots,m;\ y_1 = \frac{f_1}{a_{11}}. \qquad (6.5)$$

Рассмотрим теперь оставшиеся уравнения системы (6.3):

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{im}x_m = f_i,\quad i = 2,3,\ldots,\ m. \qquad (6.6)$$

Умножим (6.4) на a_{i1} и вычтем полученное уравнение из i-го уравнения системы (6.6), $i=2,\ldots,m$. В результате получим следующую систему уравнений:

$$\begin{aligned} x_1 + c_{12}x_2 + \ldots + c_{12}x_j + \ldots + c_{1m}x_m &= y_1, \\ a_{22}^{(1)}x_2 + \ldots + a_{2j}^{(1)}x_2 + \ldots + a_{2m}^{(1)}x_m &= f_2^{(1)}, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots &, \\ a_{m2}^{(1)}x_2 + \ldots + a_{mj}^{(1)}x_2 + \ldots + a_{mm}^{(1)}x_m &= f_m^{(1)}. \end{aligned} \qquad (6.7)$$

Здесь введены обозначения:

$$a_{ij}^{(1)} = a_{ij} - c_{1j}a_{i1},\ f_i^{(1)} = f_i - y_1a_{i1},\quad i,j = 2,3,\ldots,\ m. \qquad (6.8)$$

Матрица системы (6.7) имеет вид:

$$\begin{bmatrix} 1 & c_{12} & \cdots & c_{1m} \\ 0 & a_{22}^{(1)} & \cdots & a_{2m}^{(1)} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & a_{m2}^{(1)} & \cdots & a_{mm}^{(1)} \end{bmatrix}.$$

Для матриц такой структуры принято следующее обозначение:

$$\begin{bmatrix} 1 & \times & \cdots & \times \\ 0 & \times & \cdots & \times \\ \cdots & \cdots & \cdots & \cdots \\ 0 & \times & \cdots & \times \end{bmatrix},$$

где символами × обозначены ненулевые элементы. В системе (6.7) неизвестное x_1 содержится только в первом уравнении, поэтому в дальнейшем достаточно иметь дело с укороченной системой уравнений

$$a_{22}^{(1)}x_2+\ldots+a_{2j}^{(1)}x_2+\ldots+a_{2m}^{(1)}x_m=f_2^{(1)},$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots, \tag{6.9}$$

$$a_{m2}^{(1)}x_2+\ldots+a_{mj}^{(1)}x_2+\ldots+a_{mm}^{(1)}x_m=f_m^{(1)}.$$

Тем самым мы осуществили первый шаг метода Гаусса. Если $a_{22}^{(1)}\neq 0$, то из системы (6.9) совершенно аналогично можно исключить неизвестное x_2 и прийти к системе, эквивалентной (2) и имеющей матрицу следующей структуры:

$$\begin{array}{cccc} 1 & \times & \cdots & \times, \\ 0 & 1 & \cdots & \times, \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & \times. \end{array}$$

При этом первое уравнение системы (6.7) остается без изменения.

Аналогичным образом выполняется исключение неизвестных $x_3, x_4, \ldots, x_m$. В результате система уравнений приводится к виду:

$$\begin{aligned} x_1+c_{12}x_2+\ldots+c_{12}x_j+\ \ldots+c_{1m}x_m&=\ y_1,\\ x_2+\ldots+c_{2j}x_j+\ \ldots+c_{2m}x_m&=\ y_2,\\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots&,\\ x_{m-1}+c_{m-1,m}x_m&=\ y_{m-1},\\ x_m&=\ y_m. \end{aligned} \tag{6.10}$$

Матрица этой системы

$$C=\begin{bmatrix} 1 & c_{12} & \cdots & c_{1m} \\ 0 & 1 & \cdots & c_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{6.11}$$

содержит нули в элементах, расположенных под главной диагональю. Матрицы такого вида называются верхними треугольными матрицами.

Получение системы (8) составляет прямой ход метода Гаусса. Обратный ход заключается в нахождении неизвестных $x_1, x_2, \ldots, x_m$ из системы (8). Поскольку матрица имеет треугольный вид, можно последовательно, начиная с x_m, найти все неизвестные. Общие формулы обратного хода имеют вид

$$x_i=y_i-\sum_{j=i+1}^{m}c_{ij}x_j,\ i=m-1,\ldots,1,\ \ x_m=y_m. \tag{6.12}$$

При реализации в программе прямого хода метода Гаусса нет необходимости работать с переменными $x_1, x_2, \ldots, x_m$. Достаточно указать алгоритм, согласно которому исходная матрица A преобразуется к треугольному виду (6.11), и указать соответствующие преобразования правых частей системы. Получим эти общие формулы. Пусть реализованы первые $k-1$ шагов, т.е. уже исключены переменные $x_1, x_2, \ldots, x_{k-1}$. Тогда имеем систему

$$\begin{aligned}
x_1 + c_{12}x_2 + \ldots + c_{1k}x_k + \ldots + c_{1m}x_m &= y_1, \\
x_2 + \ldots + c_{2k}x_k + \ldots + c_{2m}x_m &= y_2, \\
&\ldots\ldots\ldots\ldots\ldots\ldots, \\
x_{k-1} + c_{k-1,k}x_k + \ldots + c_{k-1,m}x_m &= y_{k-1}, \\
a_{k,k}^{(k-1)}x_k + \ldots + a_{k,m}^{(k-1)}x_m &= f_k^{(k-1)}, \\
&\ldots\ldots\ldots\ldots\ldots\ldots, \\
a_{m,k}^{(k-1)}x_k + \ldots + a_{m,m}^{(k-1)}x_m &= f_m^{(k-1)}.
\end{aligned} \tag{6.13}$$

Рассмотрим k-е уравнение этой системы:

$$a_{k,k}^{(k-1)}x_k + \ldots + a_{k,m}^{(k-1)}x_m = f_k^{(k-1)}.$$

Предположим, что $a_{k,k}^{(k-1)} \neq 0$. Поделив обе части этого уравнения на $a_{k,k}^{(k-1)}$, получим

$$x_k + c_{k,k+1}x_k + \ldots + c_{k,m}x_m = y_k, \tag{6.14}$$

где

$$c_{kj} = \frac{a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}},\ j = k+1,\ k+2, \ldots, m,$$

$$y_k = \frac{f_k^{(k-1)}}{a_{kk}^{(k-1)}}.$$

Далее необходимо умножить уравнение (6.14) на $a_{ik}^{(k-1)}$ и вычесть полученное соотношение из i-го уравнения системы (6.13), где $i=k+1,k+2,\ldots,m$. В результате последняя группа уравнений системы (6.13) примет вид:

$$\begin{aligned}
x_k + c_{k,k+1}x_k + \ldots + c_{k,m}x_m &= y_k, \\
a_{k+1,k+1}^{(k)}x_{k+1} + \ldots + a_{k+1,m}^{(k)}x_m &= f_k^{(k)} \\
&\ldots\ldots\ldots\ldots\ldots\ldots,
\end{aligned}$$

$$a_{m,k+1}^{(k)} x_{k+1} + \ \ldots + a_{m,m}^{(k)} x_m = \ f_m^{(k)},$$

где

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} c_{kj}, \ i, j = k+1, k+2, \ldots, m,$$

$$f_i^{(k)} = f_i^{(k-1)} - a_{ik}^{(k-1)} y_k, \ i = k+1, k+2, \ldots m .$$

Таким образом, в прямом ходе метода Гаусса коэффициенты уравнений преобразуются по следующему правилу:

$$a_{kj}^{(0)} = a_{kj}, \ k, j = 1, 2, \ldots, m,$$

$$c_{kj} = \frac{a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}}, \ j = k+1, k+2, \ldots, m, \ k = 1, 2, \ldots, m . \qquad (6.15)$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} c_{kj}, \ i, j = k+1, k+2, \ldots, m, \ k = 1, 2, \ldots, m-1 .$$

Вычисление правых частей системы (6.10) осуществляются по формулам:

$$f_k^{(0)} = f_k, \ y_k = \frac{f_k^{(k-1)}}{a_{kk}^{(k-1)}}, \ k = 1, 2, \ldots, m, \qquad (6.16)$$

$$f_i^{(k)} = f_i^{(k-1)} - a_{ik}^{(k-1)} y_k, \ i = k+1, k+2, \ldots m . \qquad (6.17)$$

Коэффициенты c_{ij} и правые части y_i, $i=1,2,..m$, $j=i+1,i+2,\ldots, m$, хранятся в памяти и используются при осуществлении обратного хода по формулам (6.12).

Очевидно, что при выполнении прямого хода важным является предположение о том, что все элементы $a_{kk}^{(k-1)}$ отличны от нуля. Число $a_{kk}^{(k-1)}$ называется ведущим элементом на k-м шаге исключения. Даже если какой-то ведущий элемент не равен нулю, а просто близок к нему, в процессе вычислений может происходить сильное накопление погрешностей. Выход из этой ситуации заключается в том, что в качестве ведущего элемента выбирается не $a_{kk}^{(k-1)}$, *а другое число.*

В листинге 6.1 представлен метод *Gauss()*, который получает матрицу *A* и столбец свободных членов *B* и реализует прямой и обратных ход метода Гаусса.

Листинг 6.1. Метод Гаусса

```
public static void Gauss(float[,] A, float[] B,
```

```
                    ref float[] X)
{
   int Count = B.Length;
   // решение системы: прямой ход
   for (int i = 0; i <= Count - 2; i++)
      for (int j = i + 1; j <= Count-1; j++)
      {
         for (int k=i+1; k <= Count-1; k++)
            A[k,j] += -A[i,j]*A[k,i]/A[i,i];
            B[j] += -B[i]*A[i,j] / A[i, i];
      }
   // решение системы: обратный ход
   for (int j = Count-1; j >= 0; j--)
   {
      X[j] = B[j];
      for (int k = j + 1; k <= Count-1; k++)
         X[j] += -A[k, j] * X[k];
       X[j] /= A[j, j];
   }
}
```

6.2. Приближенное вычисление производных

Задача численного дифференцирования состоит в приближенном вычислении производных функции $u(x)$ по значениям этой функции, заданным в конечном числе точек. Пусть на $[a, b]$ введена сетка

$$w_h = \{x_i = a + ih,\ i = 0,1,\ldots N,\ hN = b - a\}$$

и определены значения $u_i = u(x_i)$ функции $u(x)$ в точках сетки. В качестве приближенного значения $u'(x_i)$ можно, например, взять любое из следующих разностных отношений:

$$u_{x,i} = \frac{u_i - u_{i-1}}{h},\ u_{x,i} = \frac{u_{i+1} - u_{i-1}}{h},\ u_{x,i} = \frac{u_{i+1} - u_{i-1}}{2h}.$$

В качестве примера рассмотрим функцию

$$f(x) = x^3,$$

производная которой равна $f'(x) = 3x^2$.

Код программы представлен в листинге 6.2.

Листинг 6.2. Приближенное вычисление производной

```
float a = 0;
float b = 3;
int m = 50;
float dx = (b - a) / m;
float x = 0, y = 0, yt = 0;
for (int i = 1; i <= m; i++)
{
   x += dx;
   y = (F3(x) - F3(x - dx)) / dx; // левая разность
   yt = (F3(x + dx) - F3(x - dx)) / (2 * dx);
   g.DrawEllipse(Pens.Black,II(x)-2,JJ(y)-2, 4, 4);
// центральная разность
   g.DrawRectangle(Pens.Black,II(x)-2,JJ(yt)-2,4,4);
}
```

Результаты расчетов представлены на рисунке 6.1.

На рисунке 6.1 квадратами представлены приближенные значения производной, вычисленной через центральные разности. Видно, что эти значения совпадают с точными значениями производной, обозначенными в соответствующих точках кружками.

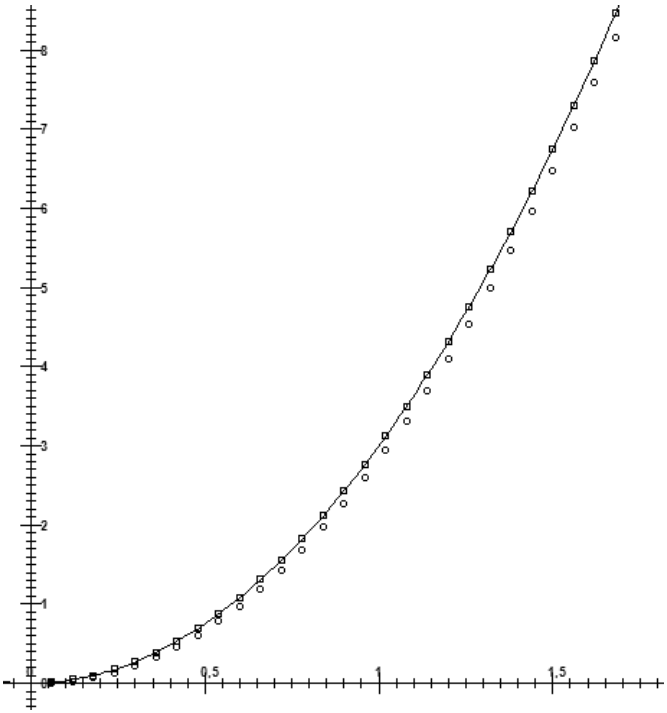


Рис. 6.1. Результаты вычисления производной функции x^3

6.3. Приближенное вычисление интегралов

Все приближенные методы вычисления определенных интегралов

$$I = \int_a^b f(x)\,dx \tag{6.18}$$

основаны на замене интеграла конечной суммой

$$I_n = \sum_{k=0}^{N} c_k f(x_k),$$

где c_k – числовые коэффициенты и x_k – точки отрезка $[a, b]$, $k = 0, 1, \ldots, N$. Приближенное равенство

$$\int_a^b f(x)\,dx \approx \sum_{k=0}^{N} c_k f(x_k) \tag{6.19}$$

называется квадратурной формулой, а сумма вида (6.19) – квадратурной суммой. Точки x_k называются узлами квадратурной формулы, а числа c_k – коэффициентами квадратурной формулы.

Введем на $[a, b]$ равномерную сетку с шагом h, т. е. множество точек

$$w_h = \{x_i = a + ih, \;\; i = 0, 1, \ldots N, \;\; hN = b - a\},$$

и представим интеграл (1) в виде суммы интегралов по частичным отрезкам:

$$\int_a^b f(x)\,dx = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)\,dx. \tag{6.20}$$

Для построения формулы численного интегрирования на всем отрезке $[a, b]$ достаточно построить квадратурную формулу для интеграла

$$\int_{x_{i-1}}^{x_i} f(x)\,dx \tag{6.21}$$

на частичном отрезке $[x_{i-1}, x_i]$ и воспользоваться свойством (6.20).

6.3.1. Формула прямоугольников

Заменим интеграл (6.18) выражением $f(x_{i-1/2})h$, где $x_{i-1/2}=x_i-h/2$. Геометрически такая замена означает, что площадь криволинейной трапеции $ABCD$ заменяется площадью прямоугольника $ABCD$ (рис. 6.2). Тогда получаем формулу

$$\int_{x_{i-1}}^{x_i} f(x)\,dx \approx f\left(x_{i-\frac{1}{2}}\right)h, \tag{6.22}$$

которая называется формулой прямоугольников на частичном отрезке $[x_{i-1}, x_i]$.

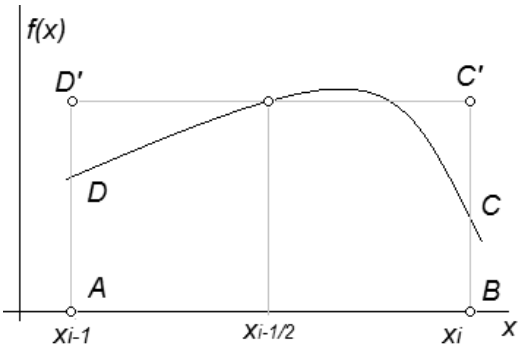


Рис. 6.2. Геометрический смысл формулы прямоугольников

Суммируя равенства (6.22) по i от 1 до N, получаем составную формулу прямоугольников

$$\int_a^b f(x)\,dx = \sum_{i=1}^{N} f\left(x_{i-\frac{1}{2}}\right)h. \qquad (6.23)$$

На рисунке 6.3 показан результат вычисления интеграла по формуле прямоугольников.

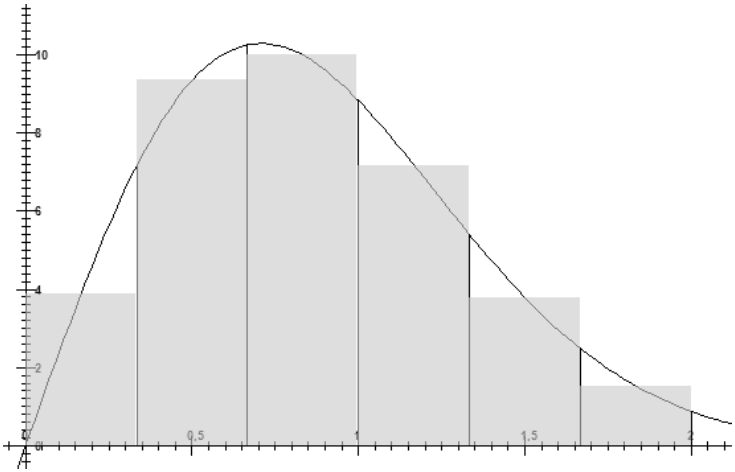


Рис. 6.3. Вычисление интеграла по формуле прямоугольников

6.3.2. Формула трапеций

На частичном отрезке эта формула имеет вид

$$\int_{x_{i-1}}^{x_i} f(x)\,dx = \frac{f(x_{i-1}+f(x_i)}{2}h. \qquad (6.24)$$

Она получается путем замены подынтегральной функции $f(x)$ интерполяционным многочленом первой степени, построенным по узлам x_{i-1}, x_i, то есть функцией

$$L_{1,i}(x) = \frac{1}{h}((x-x_{i-1})f(x_i) - (x-x_i)f(x_{i-1})).$$

Составная формула трапеций имеет вид

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} \frac{f(x_i)+f(x_{i-1})}{2}h = h\left(0{,}5f_0 + f_1 + \ldots + f_{N-1} + 0{,}5f_N\right), \qquad (6.25)$$

где

$$f_i = f\left(x_i\right),\ i = 0,1,\ldots,N, hN = b - a\ .$$

На рисунке 6.4 показан результат вычисления интеграла по формуле трапеций.

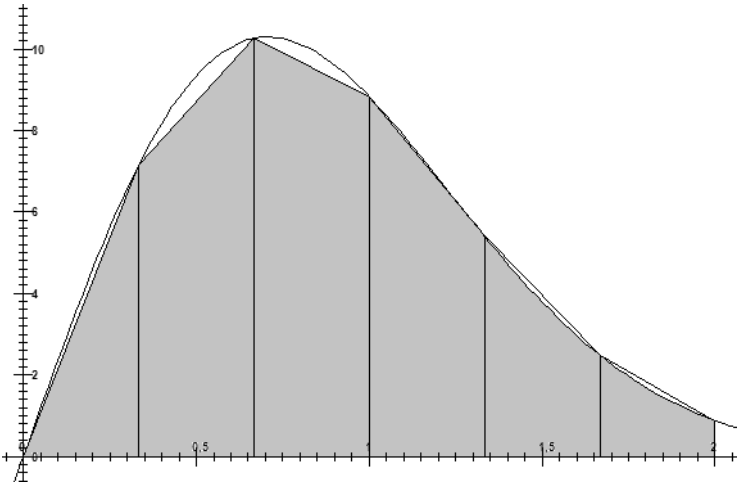


Рис. 6.4. Вычисление интеграла по формуле трапеций

6.3.3. Формула Симпсона

При аппроксимации интеграла (6.21) заменим функцию *f(x)* параболой, проходящей через точки $(x_j, f(x_j))$, $j=i-1, i-0{,}5, i$, т.е. представим приближенно *f(x)* в виде

$$f\left(x\right) \approx L_{2,i}\left(x\right),\ x \in \left[x_{i-1}, x_i\right],$$

где $L_{2,i}\left(x\right)$ – интерполяционный многочлен Лагранжа второй степени,

$$L_{2,i}\left(x\right) = \frac{2}{h^2}\left(\left(x - x_{i-\frac{1}{2}}\right)\left(x - x_i\right) f_{i-1} - 2\left(x - x_{i-1}\right)\left(x - x_i\right) f_{i-\frac{1}{2}} + \left(x - x_i\right)\left(x - x_{i-\frac{1}{2}}\right) f_i\right). \tag{6.26}$$

Проводя интегрирование, получим

$$\int_{x_{i-1}}^{x_i} L_{2,i}\left(x\right) dx = \frac{h}{6}\left(f_{i-1} + 4 f_{i-\frac{1}{2}} + f_i\right),\ h = x_i - x_{i-1}.$$

Таким образом, приходим к приближенному равенству

$$\int_{x_{i-1}}^{x_i} L_{2,i}(x)\,dx \approx \frac{h}{6}\left(f_{i-1} + 4f_{i-\frac{1}{2}} + f_i\right),$$

которое называется формулой Симпсона или формулой парабол.

На всем отрезке [a,b] формула Симпсона имеет вид

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} \frac{h}{6}\left(f_{i-1} + 4f_{i-\frac{1}{2}} + f_i\right) =$$

$$= \frac{h}{6}\left(f_0 + f_N + 2\left(f_1 + f_2 + \ldots + f_{N-1}\right) + 4\left(f_{\frac{1}{2}} + f_{\frac{3}{2}} + \ldots + f_{N-1/2}\right)\right).$$

Чтобы не использовать дробных индексов, можно обозначить

$$x_i = a + 0{,}5hi,\ f_i = f(x_i),\ i = 0,1,\ldots,2N,\ hN = b - a$$

и записать формулу Симпсона в виде

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6N}\left(f_0 + f_{2N} + 2\left(f_2 + f_4 + \ldots + f_{2N-2}\right) + 4\left(f_1 + f_3 + \ldots + f_{2N-1}\right)\right). \tag{6.27}$$

На рисунке 6.5 показан результат вычисления интеграла по формуле Симпсона.

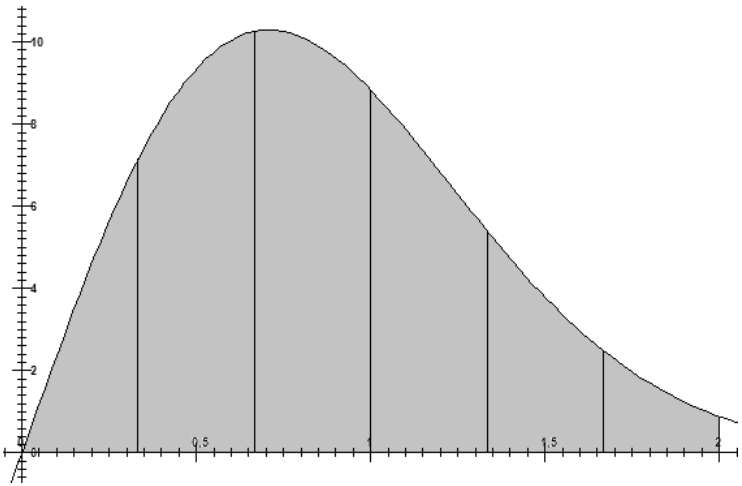


Рис. 6.5. Вычисление интеграла по формулам Симпсона

Все три приближенных способа представлены функциями *MethodRectangle(), MethodTrapezes()* и *MethodSimpson()* в листинге 6.3.

Листинг 6.3. Приближенное вычисление интеграла

```
static float F(float x)
{
   return 3 * x * x;
}
static float F1(float x)
{
   return x * x * x;
}

static float MethodRectangle(float a, float b)
{
   int n = 200;
   float dx = (b - a) / n;
   float result = 0;
   for (int i = 1; i <= n; i++)
      result += dx * F(a+i*dx);
   return result;
}

static float MethodTrapezes(float a, float b)
{
   int n = 200;
   float dx = (b - a) / n;
   float result = (F(a)+F(b))*dx/2;
   for (int i = 1+1; i <= n-1; i++)
      result += dx * F(a + i * dx);
```

```
    return result;
}

static float MethodSimpson(float a, float b)
{
    int n = 200;
    float dx = (b - a) / n;
    int[] k = new int[] { 2, 4 };
    float result = (F(a) + F(b)) * dx / 3;
    for (int i = 1 + 1; i <= n - 1; i++)
        result += k[i % 2]*dx*F(a + i * dx)/3;
    return result;
}

static void Main(string[] args)
{
    float a = 0;
    float b = 1;
    Console.WriteLine("Точное={0}",F1(b)-F1(a));

    float s = MethodRectangle(a, b);
    Console.WriteLine("Метод прямоугольников");
    Console.WriteLine("s = {0}", s);
    Console.WriteLine("");

    s = MethodTrapezes(a, b);
    Console.WriteLine("Метод трапеций");
    Console.WriteLine("s = {0}", s);
    Console.WriteLine("");
```

```
        s = MethodSimpson(a, b);
        Console.WriteLine("Метод Симпсона");
        Console.WriteLine("s = {0}", s);
        Console.WriteLine("");


        Console.ReadKey();
    }
```

В качестве примера рассмотрим интеграл

$$\int_0^1 3x^2 dx = x^3 \,|_0^1 = 1 .$$

Полученные результаты показывают следующее:

```
Точное = 1
Метод прямоугольников
s = 1,007512

Метод трапеций
s = 1,000012

Метод Симпсона
s = 0,9999992
```

6.4. Линейные дифференциальные уравнения

Рассмотрим задачу Коши для системы обыкновенных дифференциальных уравнений

$$\frac{du(t)}{dt} = f(t,u),\ t>0, \quad u(0) = u^{(0)} \tag{6.28}$$

или подробнее

$$\frac{du_i(t)}{dt} = f_i(t,u_1,u_2,\ldots,u_m),\ t>0, i=1,2,\ldots,m, \tag{6.29}$$

$$u_i(0) = u_i^{(0)},\ i=1,2,\ldots,m. \tag{6.30}$$

Существуют две группы численных методов решения задачи Коши: многошаговые разностные методы и методы Рунге-Кутта. Приведем примеры и поясним основные понятия, возникающие при использовании численных методов. Для простоты будем рассматривать одно уравнение

$$\frac{du}{dt} = f(t,u),\ t > 0, u(0) = u_0 . \qquad (6.31)$$

Введем по переменному t равномерную сетку с шагом $\tau > 0$, т.е. рассмотрим множество точек

$$\omega_\tau = \{t_n = n\tau,\ n = 0,1,\ldots\} .$$

Будем обозначать через $u(t)$ точное решение задачи (6.31), а через $y_n = y(t_n)$ – приближенное решение. Заметим, что приближенное решение является сеточной функцией, т.е. определено только в точках сетки ω_τ.

Пример 1. *Метод Эйлера*. Уравнение (6.31) заменяется разностным уравнением

$$\frac{y_{n+1} - y_n}{\tau} - f(t_n, y_n) = 0,\ n = 0,1,\ldots,\ y_0 = u_0 . \qquad (6.32)$$

Решение этого уравнения находится явным образом по рекуррентной формуле

$$y_{n+1} = y_n + \tau f(t_n, y_n),\ n = 0,1,\ldots,\ y_0 = u_0 .$$

Пример 2. Симметричная схема. Уравнение (6.31) заменяется разностным уравнением

$$\frac{y_{n+1} - y_n}{\tau} - \frac{1}{2}\left(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\right) = 0,\ n = 0,1,\ldots,\ y_0 = u_0 . \qquad (6.33)$$

Данный метод более сложен в реализации, чем метод Эйлера (6.32), так как новое решение y_{n+1} определяется по найденному ранее y_n путем решения уравнения

$$y_{n+1} - 0,5 f(t_{n+1}, y_{n+1}) = F_n , \qquad (6.34)$$

где $F_n = y_n + 0,5\tau f(t_n, y_n)$. По этой причине метод называется неявным. Преимуществом метода (6.34) по сравнению с методом (6.32) является более высокий порядок точности.

Приведенные примеры представляют собой простейшие случаи разностных методов. Методы Рунге-Кутта отличаются от разностных методов тем, что в них допускается вычисление правых частей $f(t, u)$ не только в точках сетки, но и в некоторых промежуточных точках.

Пример 3. Метод Рунге-Кутта второго порядка точности. Предположим, что известно приближенное значение y_n решения исходной задачи в

момент $t = t_n$. Для нахождения $y_{n+1} = y(t_{n+1})$ поступим следующим образом. Сначала, используя схему Эйлера

$$\frac{y_{n+1/2} - y_n}{0,5\tau} = f\left(t_n, y_n\right), \tag{6.35}$$

вычислим промежуточное значение $y_{n+1/2}$, а затем воспользуемся разностным уравнением

$$\frac{y_{n+1} - y_n}{0,5\tau} = f\left(t_n + 0,5\tau, y_{n+1/2}\right), \tag{6.36}$$

из которого явным образом найдем искомое значение y_{n+1}.

Реализация метода в два этапа (6.35), (6.36) называется методом предиктор-корректор (предсказывающе-исправляющим), т. к. на первом этапе (6.35) приближенное значение предсказывается с невысокой точностью $O(\tau)$, а на втором этапе (6.36) точность имеет второй порядок по τ.

В качестве примера рассмотрим линейное дифференциальное уравнение

$$y' = f\left(t, y\right) = y*t, \;\; y\left(0\right) = 1.$$

Это уравнение имеет точное решение $y = e^{x^2}$. Код для реализации метода Эйлера показан в листинге 6.4.

Листинг 6.4. Решение уравнения методом Эйлера

```
a = 0; b = 2; m = 50;
dx = (b - a) / m;
x = 0, y = 1;
for (int i = 1; i <= m; i++)
{
   y += dx*F(x,y);
   x += dx;
   g.DrawEllipse(Pens.Black, II(x) - 2, JJ(y) - 2,
4, 4);
}
```

Для симметричной схемы уравнение (6.31) принимает вид:

$$\frac{y_{n+1}-y_n}{\tau}-\frac{1}{2}\left(\left(t_n{*}y_n\right)+\left(t_{n+1}{*}y_{n+1}\right)\right)=0,\ n=0,1,\ldots,\ y_0=u_0$$

и его удается решить явно:

$$y_{n+1}=\frac{y_n+\tau{*}f\left(t_n,x_n\right)/2}{1-\tau/2{*}x_{n+1}}.$$

Код для решения уравнения по симметричной схеме представлен ниже.

Листинг 6.5. Решение уравнения по симметричной схеме

```
a = 0; b = 2;
m = 50;
dx=(b-a)/m;
x = 0; y = 1;
for (int i = 1; i <= m; i++)
{
   y = (y+dx/2*F(x,y))/(1-(x+dx)*dx/2);
   x += dx;
   g.FillEllipse(myBrush,II(x)-3, JJ(y)-3,6,6);
}
```

В следующем листинге 6.6 представлен код для реализации метода Рунге-Кутта:

Листинг 6.6. Решение уравнения методом Рунге-Кутта

```
a = 0; b = 2;
m = 50;
dx = (b - a) / m;
x = 0; y = 1; yt = 0;
for (int i = 1; i <= m; i++)
{
   yt = y + dx / 2 * F6(x, y);
```

```
            y = y + dx / 2 * F6(x + dx / 2, yt);
            x += dx;
            g.FillEllipse(myBrush,II(x)-2,JJ(y) - 2, 4, 4);
        }
```

На рисунке 6.6 на интервале [0, 2] представлено точное решение (сплошная линия), приближенное решение по методу Эйлера (маленькие черные точки), по симметричной схеме (большие черные точки) и методом Рунге-Кутта (белые точки).

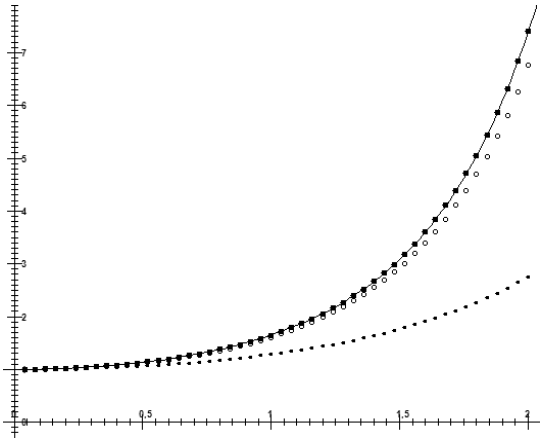


Рис. 6.6. Сравнение точного и приближенных решений линейного уравнения

Из рисунка 6.6 видно, что метод приближенного решения по симметричной схеме обеспечивает более точное решение.

Литература

1. Ford, William H., Topp, William R. Data Structures with C++ Using STL. – Upper Saddle River, New York: Prentice Hall, 2002. – 1039 p.
2. McMillan, Michael. Data Structures and Algorithms Usung C#. – New York: Cambridge University Press, 2007. – 366 p.
3. Агафонов В.Н. Типы и абстракция данных в языках программирования. Данные в языках программирования. – М.: Мир, 1982.
4. Ахо А., Холкрофт Дж., Ульман Дж. Структуры данных и алголритмы. – М. : Издательский дом «Вильямс», 2016. – 400 с. : ил.
5. Ахо А., Сети Р., Ульман Дж. Компиляторы. Принципы, технологии, инструменты. – М.: Вильямс, 2003. – 766 с.
6. Баррон Д. Введение в языки программирования. – М.: Мир, 1980.
7. Бентли Дж. Жемчужины программирования. – СПб.: Питер, 2002. – 268 с.
8. Березина Л.Ю. Графы и их применение. – М.: Просвещение, 1979. – 143 с.
9. Берж К. Теория графов и ее применения. – М.: Издательство иностранной литературы, 1962.
10. Брукс Ф.П. Как проектируются и создаются программные комплексы. – М.: Наука, 1979.
11. Виноградов М.М. Модели данных и отображения моделей данных: алгебраический подход. Теория и приложения систем баз данных. – М.: ЦЭМИ АП СССР, 1984.
12. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985.
13. Вирт Н. Алгоритмы и структуры данных. – М.: Мир, 1989, СПб.: Невский диалект, 2001. – 351 с.
14. Вирт Н. Систематическое программирование. Введение. – М.: Мир, 1982.
15. Грис Д. Наука программирования. – М.: Мир, 1984.
16. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов. – М.: Мир, 1981.
17. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. – М.: Мир, 1982.
18. Данные в языках программирования / под ред. В.Н. Агафонова. – М.: Мир, 1982.
19. Дейкстра Э. Дисциплина программирования. – М.: Мир, 1978. – 274 с.

20. Емеличев В.А., Ковалев М.М., Кравцов М.К. Многогранники, графы, опитимизация. – М.: Наука, 1981. – 341 с.
21. Емеличев В.А., Мельников О.И., Саванов В.И., Тышкевич Р.И. Лекции по теории графов. – М.: Наука, 1990.
22. Замулин А.В. Типы данных в языках программирования и базах данных. – Новосибирск: Наука, 1987.
23. Кнут Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. – М.: Мир, 1976.
24. Кнут Д. Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. – М.: Мир, 1976.
25. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Алгоритмы: построение и анализ : пер. с англ. – М.: Издательский дом «Вильямс», 2015. – 1328 с.
26. Кристофидес Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978. – 432 с.
27. Кушниренко А.Г., Лебедев Г.В. Программирование для математиков. – М.: Наука, 1988.
28. Леман Д., Смит М. Типы данных. Данные в языках программирования. – М.: Мир, 1982.
29. Ленгсам Й., Огенстайн М., Тененбаум А. Структуры данных для персональных ЭВМ. – М.: Мир, 1989.
30. Липский В. Комбинаторика для программистов. – М.: Мир, 1988. – 213 с.
31. Макаровский Б.Н. Информационные системы и структуры данных: учебное пособие для вузов. – М.: Статистика, 1980.
32. Мейер Б., Бодуэн К. Методы программирования. – М.: Мир, 1982. – 356 с.
33. Новиков Ф.А. Дискретная математика для программистов. – СПб.: Питер, 2002. – 301 с.
34. Носов В.А. Комбинаторика и теория графов. – М.: МГТУ, 1999.
35. Оре О. Теория графов. – М.: Наука, 1982 г. – 336 с.
36. Пратт Т. Языки программирования. Разработка и реализация. – М.: Мир, 1979.
37. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. – М.: Мир, 1980.
38. Седжвик, Р. Фундаментальные алгоритмы на C++ / пер. с англ. – Киев: Издательство «ДиаСофт», 2001.– 688 с.
39. Седжвик Р. Фундаментальные алгоритмы на C++. Алгоритмы на графах. — СПб.: ООО «ДиаСофтЮП», 2002.
40. Стивенс Р. Delphi. Готовые алгоритмы. – М.: ДМК Пресс; СПб.: Питер, 2004. – 384 с.
41. Трамбле Ж., Соренсон П. Введение в структуры данных. – М.: Машиностроение, 1982.

42. Уоркли Дж. Архитектура и программное обеспечение микроЭВМ. Т. 1. Структуры данных. – М.: Мир, 1984.
43. Флорес И. Структуры и управление данными. – М.: Радио и связь, 1982.
44. Фостер Дж. Обработка списков. – М.: Мир, 1974.
45. Харари Ф. Теория графов. – М.: Мир, 1973.
46. Холл П. Вычислительные структуры (введение в нечисленное программирование). – М.: Мир, 1978.
47. Хоор К. О структурной организации данных / Структурное программирование. – М.: Мир, 1975.

Оглавление

Николай Аркадиевич ТЮКАЧЕВ,
Виктор Григорьевич ХЛЕБОСТРОЕВ

C#. АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com;
196105, Санкт-Петербург, пр. Юрия Гагарина, 1, лит. А.
Тел.: (812) 412-92-72, 336-25-09.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 23.10.20.
Бумага офсетная. Гарнитура Школьная. Формат 60×90 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 14,50. Тираж 30 экз.

Заказ № 1307-20.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.