# Графы

В этой главе понятия теории графов даются с точки зрения реализации программ, уделяя особое внимание алгоритмам решения типовых задач на графах. В то же время, наряду с неформальным изложением, используются относительно строгие определения и формулировки наиболее важных теорем.

Первые задачи теории графов были связаны с решением математических развлекательных задач и головоломок, например:

* задача о кенигсбергских мостах (задача Эйлера), развитие которой привело к циклу задач об обходах графов;
* задачи о перевозках, решение которых привело к созданию эффективных методов решения транспортных задач и др.

Попытки решить сформулированную в середине XIX века задачу четырех красок привели к появлению некоторых исследований графов, имеющих теоретическое и прикладное значение. Многие результаты середины XIX века, относящиеся к теории графов, были получены при решении практических проблем. Так, Г. Кирхгоф (Густав Роберт Кирхгоф немецкий физик XIX века) при составлении полной системы уравнений для токов и напряжений в электрической схеме предложил, по существу, представлять такую схему графом и находить в нем деревья, с помощью которых выделяются линейно независимые системы контуров. А. Кэли (Артур **Кэли**  английский математик XIX века), исходя из задач подсчета числа изомеров предельных углеводородов, пришел к задачам перечисления и подсчета деревьев, обладающих заданными свойствами, и решил некоторые из них.

В XX веке задачи, связанные с графами, начали возникать не только в физике, электротехнике, химии, биологии, экономике, социологии и т. д., но и внутри математики, в таких ее разделах, как алгебра, топология, теория вероятностей, теория чисел и др. Методы этих разделов стали успешно использоваться для решения задач теории графов.

Наряду с термином *граф* в начале XX века в качестве синонимов употреблялись и другие термины, например, *карта*, *комплекс*, *диаграмма*, *сеть*, *лабиринт*.

В проблематике теории графов можно выделить направления, носящие более комбинаторный или более геометрический характер. К комбинаторным относятся, например, задачи о построении графов с заданными свойствами, задачи о подсчете и перечислении графов с фиксированными свойствами. Геометрический (топологический) характер носят, например, задачи, связанные с обходами графа, и задачи, возникающие при укладке графа на различных поверхностях.

Примером результатов геометрического направления является выделение маршрутов, содержащих все вершины или все ребра графа. Известен критерий существования обхода всех ребер графа: в связном графе цикл, содержащий все ребра и проходящий по каждому ребру только один раз, существует тогда и только тогда, когда все вершины графа имеют четные степени.

Наряду с проблемами, носящими общий математический характер, в теории графов есть специфические задачи. Например, изучаются различные свойства связности графа, исследуется строение графа по свойствам связности. При анализе надежности сетей связи, электронных схем, коммуникационных сетей возникает задача о нахождении количеств непересекающихся цепей, соединяющих различные вершины графа. Так, например, наименьшее число вершин, разделяющих две несмежные вершины графа, равно наибольшему числу непересекающихся по вершинам простых цепей, соединяющих эту пару вершин.

Найдены критерии и построены эффективные алгоритмы установления *k*-связности графа.

Характерным специфическим направлением теории графов является цикл проблем, связанных с раскраской графов, в которых изучаются разбиения множества вершин (ребер), обладающие определенными свойствами, например, смежные вершины (ребра) должны принадлежать различным множествам (вершины и ребра из одного множества окрашиваются одним цветом). Так, было доказано, что наименьшее число цветов, достаточное для раскраски ребер любого графа без петель с максимальной степенью вершин , равно , а для раскраски вершин любого графа без петель и кратных ребер достаточно () цветов.

Построены алгоритмы установления изоморфизма графов: графы, у которых степени вершин ограничены константой; графы, у которых собственные значения матрицы смежности ограничены константой; графы, допускающие вложение в поверхность ограниченного константой рода.

Приведем основные определения, используемые в 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* называют соответственно *вершинами* и *ребрами* графа *G*.

*Ориентированный граф* (орграф) *G* = (*V*, *E*) есть пара множеств, где *V* — множество вершин (узлов), *E* — множество дуг (ориентированных ребер). *Дуга* — это упорядоченная пара вершин (*v*, *w*), где вершину *v* называют началом, а *w* — концом дуги. Можно сказать, что дуга *v* → *w* ведет от вершины *v* к вершине *w*, при этом вершина *w* смежная с вершиной *v*.

Два ребра, инцидентные одной вершине, называются *смежными*; две вершины, инцидентные одному ребру, также называются *смежными*.

*Планарный граф* — граф, который может быть изображен на плоскости без пересечения ребер.

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

*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.

*Псевдограф* — граф, содержащий петли.

*Мультиграф* — это граф, в котором существует пара вершин, которая соединена более чем одним ребром (ненаправленным) либо более чем двумя дугами противоположных направлений.

*Вполне несвязный граф* — граф без ребер, другие названия: регулярный степени 0 граф, пустой граф, нуль-граф.

*Двудольный граф* (или *биграф*, или *четный граф*) — это граф *G*(*V*, *E*), такой что множество *V* разбито на два непересекающихся подмножества *V*1 и *V*2, причем каждое ребро *E* инцидентно вершине из *V*1 и вершине из *V*2 (т. е. соединяет вершину из *V*1 с вершиной из *V*2). Множества *V*1 и *V*2 называются *долями* двудольного графа.

*Упорядоченный граф* — граф, в котором дуги, выходящие из каждой вершины, однозначно пронумерованы, начиная с 1. Дуги считаются упорядоченными в порядке возрастания номеров. При графическом представлении часто дуги считаются упорядоченными в порядке некоторого стандартного обхода (например, слева направо).

*Степень вершины* (валентность вершины) *v* — это количество ребер, инцидентных этой вершине *v*. Обозначается как *deg*(*v*). Минимальная степень вершины графа *G* обозначается δ(*G*), а максимальная — Δ(*G*).

*Маршрут* (или *путь*) в графе *G* — это чередующаяся последовательность вершин и ребер *v*0, *e*1, *v*1, *e*2, *v*2, ... , *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, ... , *v*t − 1, *e*t, *v*t + 1, в которой *e*i= *v*i − 1*v*i (1 ≤ *i* ≤ *t*). Такой маршрут кратко называют (*v*0, *v*t)-*маршрутом* и говорят, что он *соединяет* *v*0 c *v*t; а вершины *v*0, *v*t — *концевые* вершины указанного маршрута.

*Цепь в графе* — маршрут, все ребра которого различны.

Если все вершины (а следовательно, и ребра) различны, то такая цепь называется *простой*. В цепи *v*0, *e*1, ... , *e*k, *v*k вершины *v*0 и *v*k называются *концами* цепи. Цепь с концами *u* и *v* соединяет вершины *u* и *v*. Для орграфов цепь называется путем.

*Цикл* в орграфе — это простой путь длиной не менее 1, который начинается и заканчивается в одной и той же вершине. Простой цикл орграфов называется контуром.

*Гамильтоновой цепью* графа называется его простая цепь, которая проходит через каждую вершину графа точно один раз. Цикл графа, проходящий через каждую его вершину, называется *гамильтоновым* *циклом*. Граф называется *гамильтоновым*, если он обладает гамильтоновым циклом.

*Висячая вершина* — вершина, степень которой равна 1 (т. е. *deg*(*v*) = 1).

*Изолированная вершина* — вершина, степень которой равна 0 (т. е. *deg*(*v*) = 0).

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

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*.

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

*Длина маршрута* — количество ребер в маршруте (с повторениями). Если маршрут *M* = (*v*0, *e*1, *v*1, *e*2, *v*2, ... , *e*k, *v*k), то длина *M* равна *k* (обозначается как |*M*| = *k*). Заметим, что в обыкновенном графе маршрут полностью определяется последовательностью *v*0, *v*1, ... , *v*t своих вершин. Если *v*0 = *v*t, то (*v*0, *v*t)-маршрут называется *замкнутым*.

*Длина пути* — это количество дуг, составляющих путь. Так, для пути *v*1, *v*2, ... , *v*n длина равна (*n* − 1).

*Множество смежности* вершины *v* — множество вершин, cмежных с вершиной *v*. Обозначается как Γ + (*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).

*Регулярный граф* — граф, степени всех вершин которого равны. Степень регулярности является инвариантом графа *G* и обозначается *r*(*G*). Для нерегулярных графов *r*(*G*) не определено.

*Раскраска графа* — разбиение вершин на множества (называемые *цветами*), при котором нет двух смежных вершин, принадлежащих одному и тому же множеству.

*Размеченный граф* — это граф, для которого задано множество меток *S*, функция разметки вершин *f*: *A* → *S* и функция разметки дуг *g*: *R* → *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*.

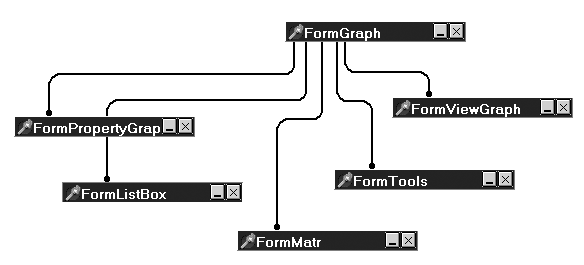
Две вершины в графе *связаны*, если существует соединяющая их простая цепь.

*Связный граф* — граф, в котором все вершины связаны.

*Хроматическое число* графа — это минимальное количество цветов, требуемое для раскраски графа.

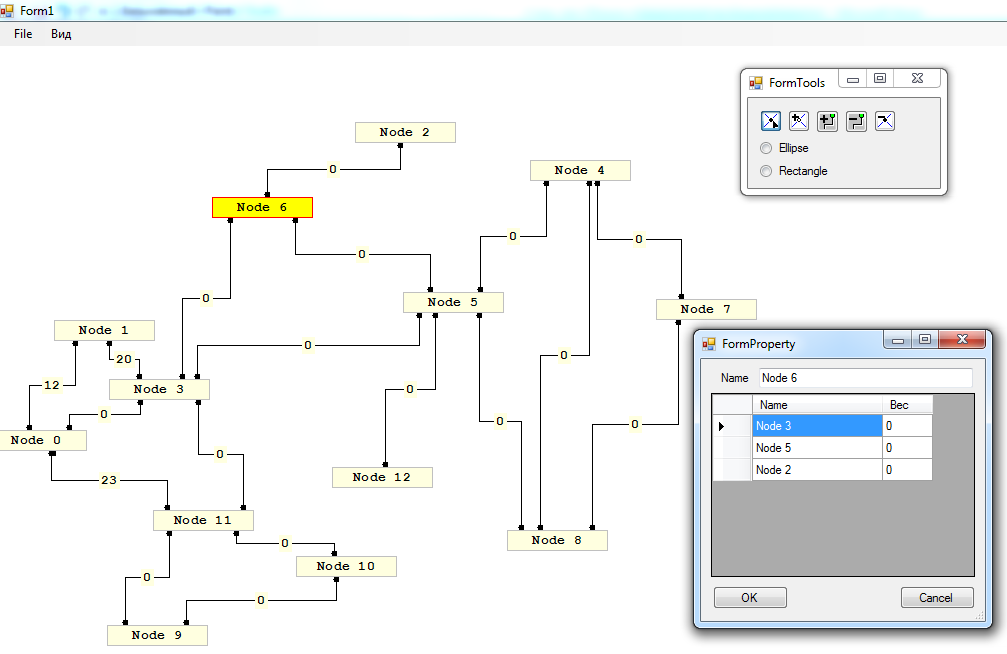
## 14.1. Проект для алгоритмов на графах

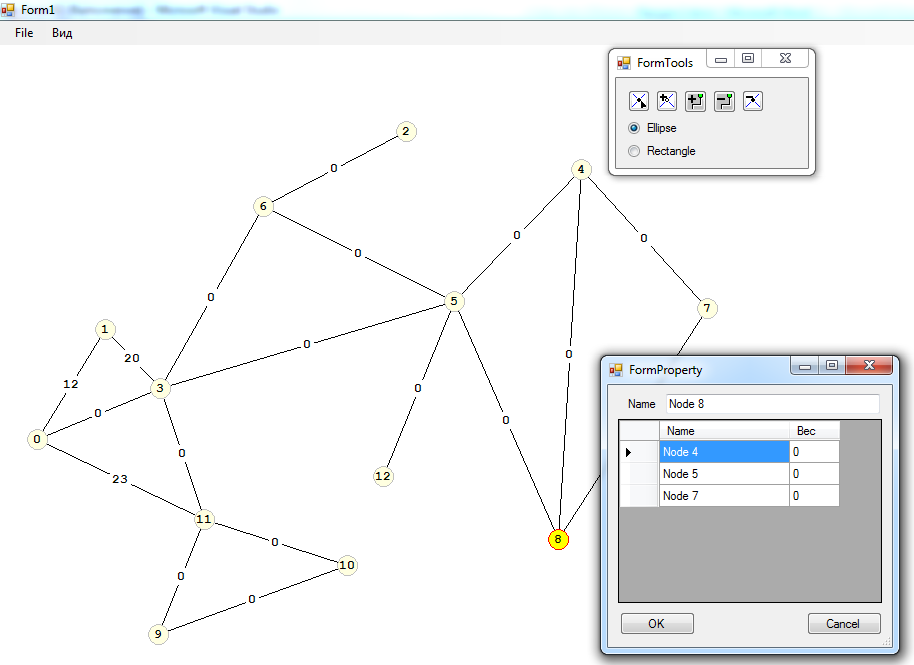
Нам придется несколько забежать вперед, обсуждая проект для реализации алгоритмов на графах, т. к. для этих алгоритмов удобен графический интерфейс. Впрочем, в проекте использованы минимальные графические возможности. Итак, проект состоит из главной формы FormGraph, шести вспомогательных форм FormPropertyGraph, FormListBox, FormMatr, FormTools, FormViewGraph и модуля LibGraph, в котором собраны структуры данных и все алгоритмы для графов (рис. 14.1).



**Рис. 14.1.** Структура проекта

Рисование графа реализовано на канве формы FormGraph и позволяет изображать граф в двух видах: в виде прямоугольников, соединенных ломаными линиями (рис. 14.2); в виде окружностей, соединенных прямыми линиями (рис. 14.3). Переключение типа графа происходит на форме FormTools. На форме FormTools находятся пять инструментальных кнопок, позволяющие добавлять, удалять и перемещать узлы, создавать и удалять дуги. Форма FormMatr показывает матрицу смежности, а с помощью формы FormProperty можно задавать свойства любого узла.



**Рис. 14.2.** Проект для алгоритмов на графах, узлы — прямоугольники**Рис. 14.3.** Проект для алгоритмов на графах, узлы — окружности

### 14.1.1. Стек

Многие алгоритмы на графах требуют помещать данные в стек или в очередь *(см. гл. 7 в [1])*. 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 выбрали наиболее простой механизм этих структур, основанный на массиве, объединив в одном классе и стек, и очередь (листинг 14.3).

Листинг 14.1. 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 стека

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 14.2. Класс узла

public class Node // узел

{

public Node next;

public object data;

public Node(Node next, object data)

// конструктор

{

this.next = next;

this.data = data;

}

}

а поля top и tail показывают на начало стека и конец очереди.

Для реализации механизмов работы со стеками и очередями введем две переменные myStack, path.

Основные операции над стеком — включение нового элемента (*push* — заталкивать) и исключение элемента из стека (*pop* — выскакивать).

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 14.3).

Листинг 14.3. Инициализация стека/очереди

public MyStack() // конструктор

{

top = null; tail = null;

}

При проверке пустоты стека необходимо проверить, что вершина стека показывает на null (листинг 14.4).

Листинг 14.4. 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 14.5).

Листинг 14.5. 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 14.6).

Листинг 14.6. 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 next конца списка направляем на новый элемент и перемещаем указатель на конец списка на новый элемент (листинг 14.7).

Листинг 14.7. 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;

}

}

### 14.1.2. Структура данных

Выбор структуры данных оказывает решающее значение на эффективность алгоритмов. Общая структура классов проекта представлена на рис. 14.4.

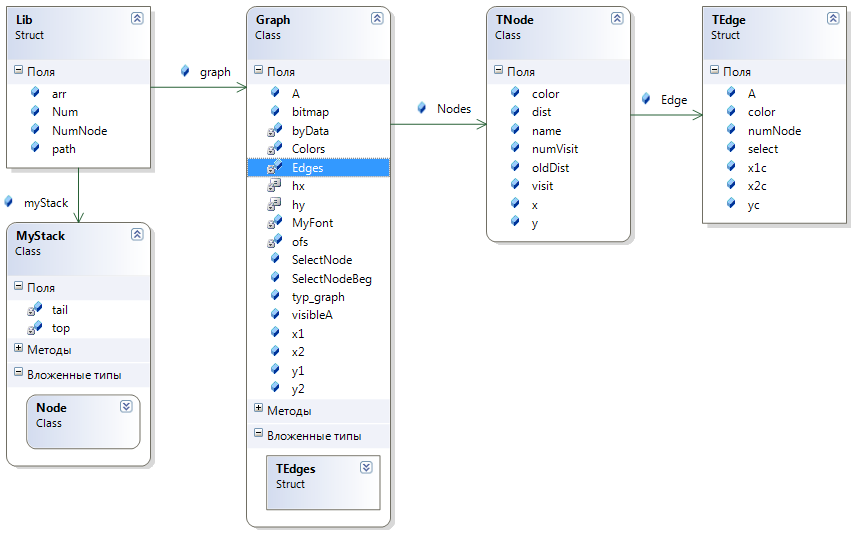


Рис. 14.4.

Класс Graph является основным

Листинг 14.8. 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 14.1.

Листинг 14.9. 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 14.10 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 14.10. Структура ребер

public struct TEdge

{

public int A; //

public int numNode; //

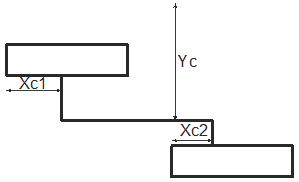
public int x1c, x2c, yc; //

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. 14.5). Поле указывает на расстояние горизонтальной части ребра от верхнего края формы.

****

**Рис. 14.5.** Геометрические параметры ребра

### 14.1.3. Изображение графов

Процедура рисования графа похожа на рисование дерева в проекте, описанном в *разд. 13.2.1*. На канву Bitmap мы сначала выводим дуги, а затем все узлы (листинг 14.9).

Листинг 14.11. 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),

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 перетаскивания узлов и создания новых дуг реализованы обработчики событий onMouseDown, onMouseMove и onMouseUp.

### 14.1.4. 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 14.12. 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 14.13. Вычисление длины байтового массива

protected int LengthFile()

// вычислить размер файла

{

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 14.14. 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:

1. перемещаем длину строки в основной байтовый массив byData;
2. перемещаем строку в символьный массив charData[];
3. с помощью класса Encoder перемещаем символьный массив во вспомогательный байтовый массив byByte[];
4. вставляем вспомогательный байтовый массив byByte[] в основной байтовый массив byData.

Листинг 14.15. 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 14.11):

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 14.16. 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 14.17. 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 14.18. 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:

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

## 14.2. Поиск в графах

Основой поиска нужной вершины в графе, как правило, является систематический перебор вершин графа, такой, что каждая вершина просматривается не более одного раза. Критериями эффективности любого алгоритма и, в частности, алгоритмов поиска являются:

* легкая "читаемость" алгоритма;
* анализ любого ребра или вершины происходит не более одного раза.

### 14.2.1. Поиск в глубину

Одним из основных алгоритмов поиска в неориентированном графе является поиск в *глубину* (*depth first search*). Общая идея этого метода заключается в следующем: начиная поиск с какой-то вершины n, просматриваем список ее дуг и, если находится соседняя непосещенная вершина, то переходим в нее, далее продолжаем алгоритм из этой вершины. Естественно, мы должны отмечать каждую посещенную вершину, меняя поле Node[n].Visit = True.

Наиболее простой случай задачи о кратчайших путях — когда все веса равны 0 или бесконечны. Другими словами: на ориентированном графе найти узлы, доступные из данного без учета весов ребер. Нас интересует другая задача: не просто перечислить все вершины, доступные из данной, но сделать это в определенном порядке.

#### 14.2.1.1. Алгоритм обхода ориентированного графа с поиском в глубину

Пусть *G* — неориентированный связный граф. В процессе поиска в глубину вершинам графа *G* присваиваются номера посещения numVisit, а его ребра помечаются.

В начале ребра не помечены, вершины не имеют номеров посещения, т. е. numVisit = 0.

Начинаем с произвольной вершины v0, присваиваем ей номер посещения: numVisit(v0) = 1, и выбираем произвольное ребро (v0, w). Ребро (v0, w) помечается как "прямое", а вершина W, достигнутая из V0, получает номер посещения numVisit(W) = 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. Далее действуем в зависимости от ситуации, следующим образом.

* Имеется непомеченное ребро (X, Y). Если у вершины Y уже есть номер посещения, то ребро (X, Y) помечаем как "обратное" и продолжаем поиск непомеченного ребра, инцидентного вершине X. Если вершина Y номера посещения не имеет, то присваиваем numVisit(Y) = Num + 1, ребро (X, Y) помечаем как "прямое" и переходим в вершину Y. Вершина Y считается получившей свой номер посещения из вершины. На следующем шаге начинаем просматривать ребра, инцидентные вершине Y.
* Все ребра, инцидентные вершине X, помечены. В этом случае возвращаемся в вершину, из которой X получила свой номер посещения.
* Процесс закончится, когда все ребра будут помечены и произойдет возвращение в вершину V0.

Описанный процесс можно реализовать так, чтобы время работы соответствующего алгоритма составляло *O*(*M* + *N*), где *M* — число ребер графа, *N* — число вершин графа.

Пусть граф *G* задан списками смежности, т. е. NV — список вершин, инцидентных вершине V, V0 — исходная вершина, с которой начинается поиск. В процессе работы алгоритма каждая вершина графа ровно один раз включается в список Q и исключается из него. Вершина включается в этот список сразу после получения номера посещения, и исключается, как только произойдет возвращение из этой вершины. Включение и исключение вершин производится всегда с конца списка, т. е. Q — стек. Результат работы алгоритма — четыре списка: список посещений NumVisit, F, T, B, где NumVisit(V) — номер посещения вершины V; F(V) — имя вершины, из которой вершина V получила свой номер посещения; T и B — это соответственно списки ориентированных "прямых" и "обратных" ребер графа *G*. Эти ребра 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.

#### 14.2.1.2. 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.

Алгоритм состоит из следующих шагов.

В начале работы алгоритма зададим эти параметры: Num = 1; numVisit(V0) = Num; Q(k) = V0; F(V0) = 0; T = 0; B = 0; p = 1.

1. V = Q(p).
2. Просматривая список NV, найти такую вершину W, что ребро (V, W) не помечено, и перейти к шагу 4. Если таких вершин нет, то перейти 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 NV. Иначе выполнить присваивания: Num := Num + 1; NumVisit(W) := Num; F(W) := V, ребро (V, W) пометить как "прямое" и занести в список T, p := p + 1; Q(p) := W — вершина получила номер посещения и занесена в стек Q. Перейти к шагу 2.
4. p := p - 1 — вершина V вычеркнута из Q. Если p = 0, то конец 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.

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 смежную с x вершину y. Если y — конечная вершина, то путь найден.
4. Проверяем наличие метки у 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**. Другой вариант алгоритма поиска в глубину основан на представлении ориентированного графа как дерева.

Пусть есть ориентированный граф, одна из вершин которого выделена. Будем полагать, что все вершины доступны из выделенной вершины по ориентированным путям. Построим дерево, которое можно было бы назвать "универсальным накрытием" нашего графа. Его корнем будет выделенная вершина графа. Из корня выходят те же стрелки, что и в графе — их концы будут сыновьями корня. Из них в дереве выходят те же стрелки, что и в графе, и т. д. Разница между графом и деревом в том, что пути в графе, ведущие в одну и ту же вершину, в дереве "расклеены". В других терминах вершина дерева — это путь в графе, выходящий из корня, а ее сыновья — это пути, продолженные на одно ребро. Заметим, что дерево бесконечно, если в графе есть ориентированные циклы.

Имеется естественное отображение дерева в граф (вершин в вершины). При этом каждая вершина графа имеет столько прообразов, сколько путей в нее ведет. Поэтому обход дерева (посещение его вершин в том или ином порядке) одновременно является и обходом графа, только каждая вершина посещается многократно.

Будем предполагать, что для каждой вершины графа выходящие из нее ребра упорядочены (например, пронумерованы). Таким образом, для каждой вершины дерева выходящие из нее ребра тоже упорядочены. Будем обходить дерево так: сначала корень, а потом поддеревья (в порядке нумерации ведущих в них ребер). Такой обход дерева рассматривался в *гл. 13*. Ему соответствует обход графа. Если выкинуть из этого обхода повторные посещения уже посещенных вершин, то и получится "поиск в глубину".

Другими словами: на путях, выходящих из выделенной вершины, введем порядок: путь предшествует своему продолжению; если два пути расходятся в некоторой вершине, то меньшим считается тот, который выходит из нее по меньшему ребру. Вершины теперь упорядочиваются в соответствии с минимальными путями, ведущими в них. Обход вершин графа в указанном порядке называется *поиском в глубину*.

Запишем этот алгоритм более подробно в терминах псевдоязыка (листинг 14.19).

Листинг 14.19. 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*-ого узла нам придется просмотреть не более *Mi* дуг. Поэтому сложность алгоритма не более *N* + Σ*Mi* = *N* + *M*, т. е. *O*(*N* + *M*).

В листинге 14.20 приведен полный текст рекурсивной процедуры поиска узла по имени nameNode в глубину.

Листинг 14.20. 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;

}

Перед первоначальным вызовом рекурсивной процедуры

// Поиск в глубину

public int DepthSearch(int n, string nameNode)

{

ClearVisit();

int result = FindDepth(n, nameNode);

return result;

}

мы должны для каждой вершины поле visit установить в значение false. Это делает процедура ClearVisit() (листинг 14.21). Попутно, эта процедура также обнуляет поле номера визита numVisit и назначает всем дугам серый цвет.

Листинг 14.21. 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 процедура VisitTrue() (листинг 14.22). Попутно эта процедура назначает вершине очередной номер визита.

Листинг 14.22. 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 14.23.

Листинг 14.23. Алгоритм нерекурсивного поиска в глубину

function DepthSearchStack(v, NameNode)

begin

CTEK := ∅;

v ⇒ CTEK;

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 ⇒ CTEK;

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 14.24 приведен полный текст нерекурсивной процедуры поиска в глубину.

Листинг 14.24. 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 14.25 содержит текст функции FindNotVisit(), предназначенной для поиска непосещенного узла.

Листинг 14.25. 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;

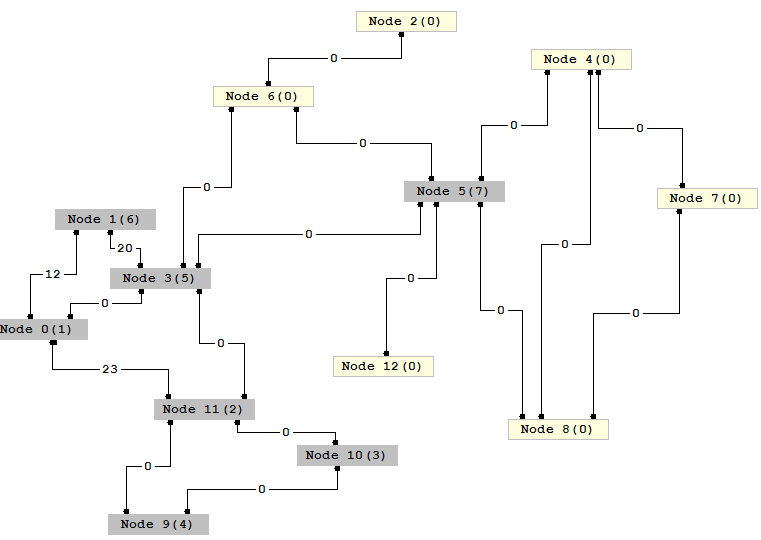
if (Ok)

result = Nodes[t].Edge[i].numNode;

return result;

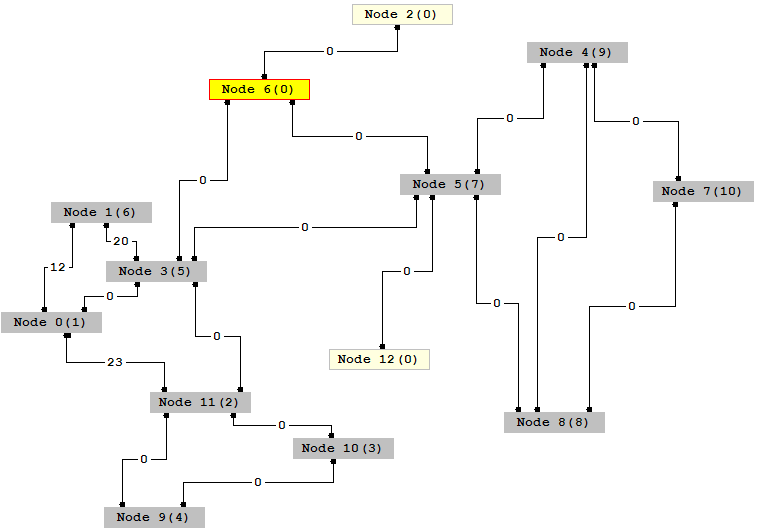
}

На рис. 14.6 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. 14.6.** Результат поиска в глубину

На рис. 14.6 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 рис. 14.7 представлен результат работы программы при поиске узла Node7.



**Рис. 14.7.** 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.

### 14.2.2. 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 14.26).

Листинг 14.26. 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 14.27 приведен полный текст нерекурсивной процедуры поиска в ширину.

Листинг 14.27. 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

}

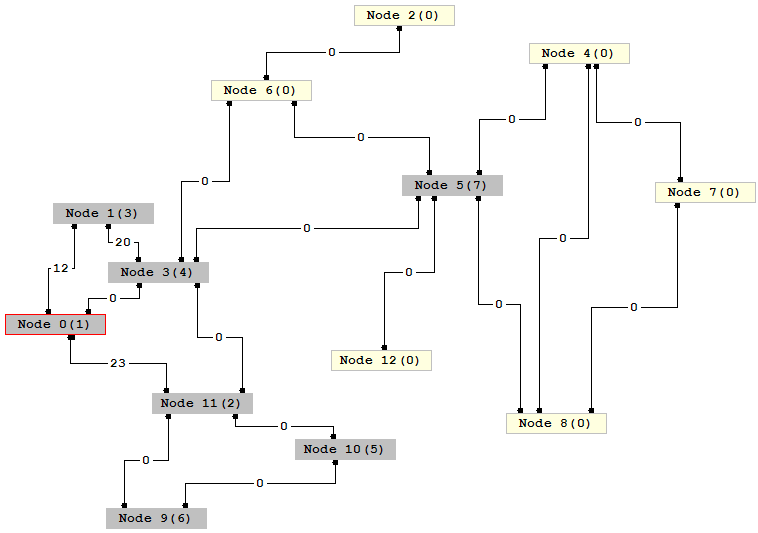
}

}

return result;

}

На рис. 14.8 приведен результат поиска в ширину от вершины Node1 до вершины Node6. Для каждого узла рядом с именем в скобках указан номер визита при поиске.



**Рис. 14.8.** Результат поиска в ширину

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.