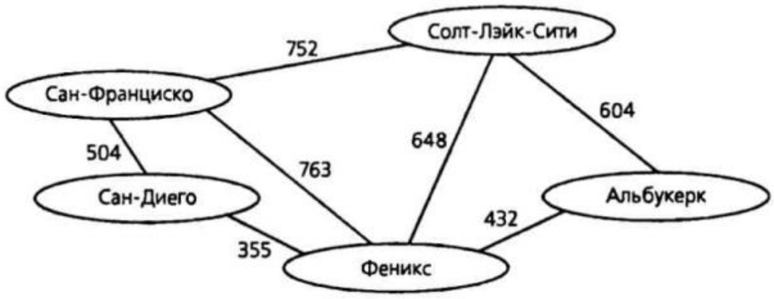
13.8. Графы

Дерево есть иерархическая структура, которая состоит из узлов, исходящих от корня. Узлы соединяются указателями от родителя к сыновьям. В этом разделе мы познакомимся с графами, которые являются обобщенными иерархическими структурами. Граф состоит из множества элементов данных, называемых **вершинами (vertices)**, и множества **ребер (edges)**, соединяющих эти вершины попарно. Ребро E = (Vi, Vj) соединяет вершины Vi и Vj.

Вершины = $\{V_0, V_1, V_2, V_3, \ldots, V_{m-1}\}$
Ребра = $\{E_0, E_1, E_2, E_3, \ldots, E_{n-1}\}$

Пусть вершины обозначают города, а ребра — дорожное сообщение между ними. Движение по дорогам может происходить в обоих направлениях, и поэ-



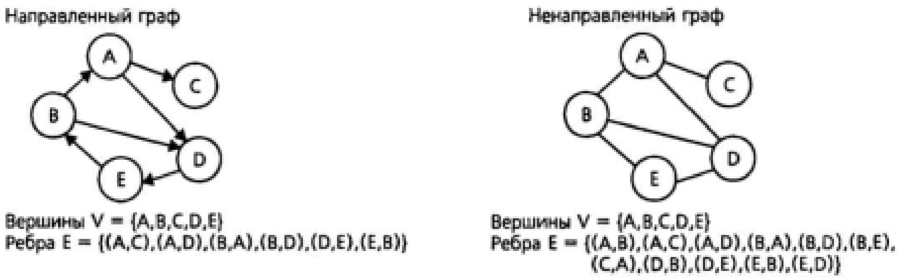


Рис.13.6. Направленный и ненаправленный графы

тому ребра графа G не имеют направлений. Такой граф называется **ненапрвленным (undirected graph)**.

Если ребра представляют систему связи с однонаправленными информационными потоками, то граф в этом случае становится **направленным графом (directed graph)**, или **орграфом (digraph)**. На рис. 13.6 показаны графы обоих типов. Мы сосредоточим внимание на орграфах.

В орграфе ребро задается парой (V_i, V_j), где V_i — начальная вершина, а V_j — конечная вершина. **Путь (path)** $P(V_S, V_E)$ есть последовательность вершин $V_S=V_R, V_{R+1}, \ldots, V_{R+T}=V_E$, где V_S — начальная вершина, V_E — конечная вершина, а каждая пара членов последовательности есть ребро. В орграфе указывается направленный путь от V_B к V_E, но пути от V_E к V_B может и не быть. Например, для орграфа на рис. 13.6

```
Путь(A,B) = {A,D,E,B}
Путь(E,C) = {E,B,A,C}
Путь(B,A) = {B,A}
Путь(C,E) = {}          // пути нет
```

Связанные компоненты

С понятием пути связано понятие связанности орграфа. Две вершины V_i и V_j **связаны (connected)**, если существует путь от V_i к V_j. Орграф является **сильно связанным (strongly connected)**, если в нем существует направленный путь от любой вершины к любой другой. Орграф является **слабо связанным (weakly connected)**, если для каждой пары вершин V_i и V_j существует направленный путь $P(V_i, V_j)$ или $P(V_j, V_i)$. Связанность графов иллюстрируется на рис. 13.7.

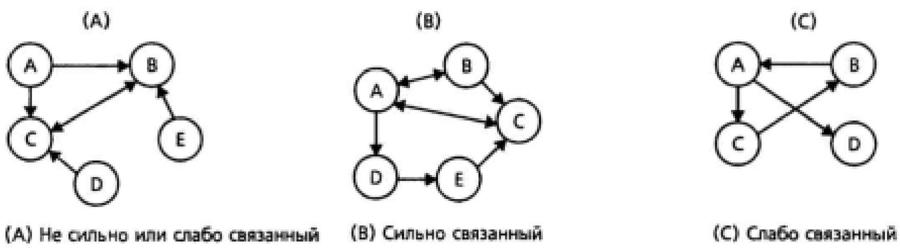
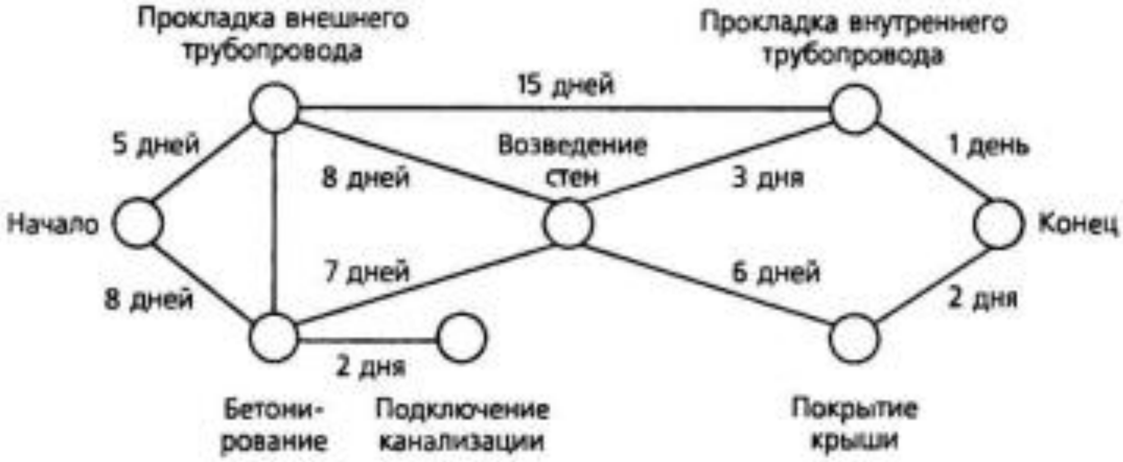


Рис. 13.7. Сильно и слабо связанные компоненты орграфа

Мы расширили понятие сильносвязанных вершин до **сильно связанной компоненты (strongly connected component)** — максимального множества вершин $\{V_i\}$, где для каждой пары V_i и V_j существует путь от V_i к V_j и путь от V_j к V_i. **Цикл (cycle)** — это путь, проходящий через три или более вершины и связывающий некоторую вершину саму с собой. В ориентированном графе (C) на рис. 13.7 существуют циклы для вершин A (A–>C–>B–>A), B и C. Граф, не содержащий циклов, называется **ациклическим (acycle)**.

Во **взвешенном орграфе (weighted digraph)** каждому ребру приписано значение, или вес. На транспортном графе веса могут представлять расстояния между городами. На графе планирования работ веса ребер определяют продолжительность конкретной работы.



13.9. Класс Graph

В этом разделе мы опишем структуру данных для взвешенного орграфа. Начнем с математического определения графа как основы абстрактного типа данных (ADT) Graph. Вершины задаются в виде списка элементов, а ребра — в виде списка упорядоченных пар вершин.

Объявление абстрактного типа данных Graph

Взвешенный орграф состоит из вершин и взвешенных ребер. ADT включает в себя операции, которые будут добавлять или удалять эти элементы данных. Для каждой вершины V_i определяются все смежные с ней вершины V_j, которые соединяются с V_i ребрами $E(V_i, V_j)$.

```
ADT Graph
   Данные
      Множество вершин {Vi} и ребер {Ei}. Ребро есть пара (Vi, Vj), которая
      указывает на связь вершины Vi с вершиной Vj. Приписанный каждому ребру
      вес определяет стоимость прохождения по этому ребру.
   Операции
      Конструктор
         Вход:          Нет
         Обработка:     Создает граф в виде множества вершин и ребер.
      InsertVertex
         Вход:          Новая вершина.
```

Предусловия: Нет
Обработка: Вставить новую вершину в множество вершин.
Выход: Нет
Постусловия: Список вершин увеличивается.

InsertEdge

Вход: Пара вершин V_i и V_j с весом W.
Предусловия: V_i и V_j должны принадлежать множеству вершин, а ребро (V_i, V_j) не должно принадлежать множеству ребер.
Обработка: Вставить ребро (V_i, V_j) с весом W в множество ребер.
Выход: Нет
Постусловия: Множество ребер увеличивается.

DeleteVertex

Вход: Ссылка на вершину V_D.
Предусловия: Входная вершина должна принадлежать множеству вершин.
Обработка: Удалить вершину V_D из списка вершин. Удалить все входящие и исходящие ребра этой вершины.
Выход: Нет
Постусловия: Множество вершин и множество ребер модифицируются.

DeleteEdge

Вход: Пара вершин V_i и V_j.
Предусловия: Входные вершины должны принадлежать множеству вершин.
Обработка: Если ребро (V_i, V_j) существует, удалить его из множества ребер.
Выход: Нет
Постусловия: Множество ребер модифицируется.

GetNeighbors

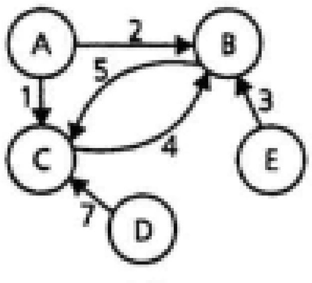
Вход: Вершина V.
Предусловия: Нет
Обработка: Идентифицировать все смежные с V вершины V_E, такие, что (V, V_E) есть ребро.
Выход: Список смежных вершин.
Постусловия: Нет

GetWeight

Вход: Пара вершин V_i и V_j.
Предусловия: Входные вершины должны принадлежать множеству вершин.
Обработка: Выдать вес ребра (V_i, V_j), если оно существует.
Выход: Вес ребра или 0, если ребра не существует.
Постусловия: Нет

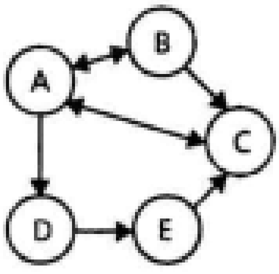
Конец ADT Graph

Представление графов. Существует много способов представления орграфов. Можно просто хранить вершины в виде последовательного списка V_0, V_1, ..., V_{m-1}, а ребра задавать квадратной матрицей размером m × m, называемой **матрицей смежности (adjcency matrix)**. Здесь строка i и столбец j соответствуют вершинам V_i и V_j. Каждый элемент (i, j) этой матрицы содержит вес ребра E_{ij} = (V_i, V_j) или 0, если такого ребра нет. Для невзвешенного орграфа элементы матрицы смежности содержат 0 или 1, показывая отсутствие или наличие соответствующего ребра. Ниже приводятся примеры орграфов со своими матрицами смежности.



$$\begin{matrix} 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \end{matrix}$$

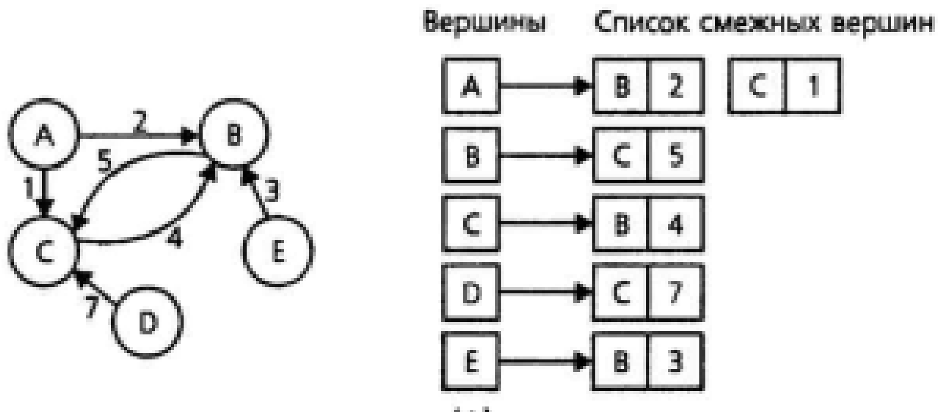
(A)



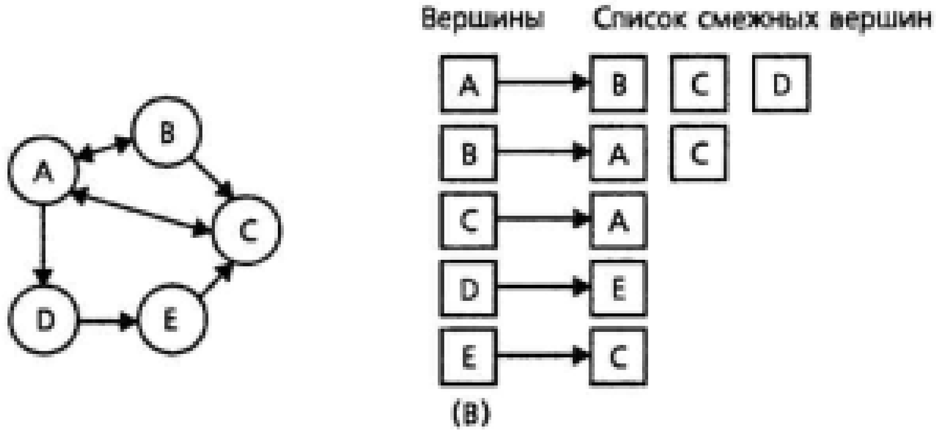
$$\begin{matrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{matrix}$$

(B)

В другом способе представления графов каждая вершина ассоциируется со связанным списком смежных с ней вершин. Эта динамическая модель хранит информацию лишь о фактически принадлежащих графу вершинах. Для взвешенного орграфа каждый узел связанного списка содержит поле веса. Примеры спискового представления орграфов даны ниже.



(A)



(B)

Класс Graph, рассматриваемый в этом разделе, использует матричное представление ребер. Мы используем статическую модель графа, которая предполагает конечное число вершин. Матричное представление упрощает реализацию класса и позволяет сосредоточиться на целом ряде алгоритмов обработки графов. Реализация на основе связанных списков предлагается в упражнениях. Основными особенностями класса Graph являются представление ADT Graph, метод ReadGraph и ряд поисковых алгоритмов, осуществляющих прохождение вершин способами "сначала в глубину" и "сначала в ширину". Данный класс включает также итератор списка вершин для использования в приложениях.

Спецификация класса Graph

ОБЪЯВЛЕНИЕ

```
const int MaxGraphSize = 25;

template <class T>
class Graph
{
  private:
    // основные данные включают список вершин, матрицу смежности
    // и текущий размер (число вершин) графа
    SeqList<T> vertexList;
    int edge [MaxGraphSize];
    int graphsize;

    // методы для поиска вершины и указания ее позиции в списке
    int FindVertex(SeqList<T> &L, const T& vertex);
    int GetVertexPos(const T& vertex);

  public:
    // конструктор
    Graph(void);

    // методы тестирования графа
    int GraphEmpty(void) const;
    int GraphFull(void) const;

    // методы обработки данных
    int NumberOfVertices(void) const;
    int NumberOfEdges(void) const;
    int GetWeight(const T& vertex1, const T& vertex2);
    SeqList<T>& GetNeighbors(const T& vertex);

    // методы модификации графа
    void InsertVertex(const T& vertex);
    void InsertEdge(const T& vertex1, const T& vertex2, int weight);
    void DeleteVertex(const T& vertex);
    void DeleteEdge(const T& vertex1, const T& vertex2);

    // утилиты
    void ReadGraph(char *filename);
    int MinimumPath(const T& sVertex, const T& sVertex);
    SeqList<T>& DepthFirstSearch(const T& beginVertex);
    SeqList<T>& BreadthFirstSearch(const T& beginVertex);

    // итератор для обхода вершин
    friend class VertexIterator<T>;
};
```

ОПИСАНИЕ

Данные-члены класса включают вершины, хранящиеся в виде последовательного списка, ребра, представленные двумерной целочисленной матрицей смежности, и переменную graphsize, являющуюся счетчиком вершин. Значение graphsize возвращается функцией NumberOfVertices.

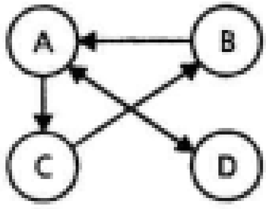
Утилита FindVertex проверяет наличие вершины в списке L и используется в поисковых методах. Метод GetVertexPos вычисляет позицию вершины vertex в vertexList. Эта позиция соответствует индексу строки или столбца в матрице смежности.

Методу ReadGraph передается в качестве параметра имя файла с входным описанием вершин и ребер графа.

Класс VertexIterator является производным от класса SeqListIterator и позволяет осуществлять прохождение вершин. Итератор упрощает приложения.

ПРИМЕР

```
Graph <char> G;                      // граф с символьными вершинами
G.ReadGraph("graph.dat");            // ввести данные из graph.dat
// Пример входного описания графа
<Количество вершин>   4
Вершина0              A
Вершина1              B
Вершина2              C
Вершина3              D
<Количество ребер>    5
Ребро0     Вес0       A C 1
Ребро1     Вес1       A D 1
Ребро2     Вес2       B A 1
Ребро3     Вес3       C B 1
Ребро4     Вес4       D A 1
```




```
VertexIterator<char> viter(G);          // итератор для вершин
SeqList<char>L;

for (viter.Reset(); !viter.EndOfList(); viter.Next())
{
  cout << "Вершины, смежные с вершиной " << viter.Data() << ": ";

  L = G.GetNeighbors(viter.Data());

  // распечатать смежные вершины
  SeqListIterator<char> liter(L);       // список смежных вершин
  for (liter.Reset(); !liter.EndOfList(); liter.Next())
    cout << liter.Data() << " ";
}
```

Реализация класса Graph

Конструктор класса Graph "отвечает" за инициализацию матрицы смежности размера MaxGraphSize × MaxGraphSize и обнуление переменной graphsize. Конструктором обнуляется каждый элемент матрицы для указания на отсутствие ребер.

```
// конструктор. обнуляет матрицу смежности и переменную graphsize
template <class T>
Graph<T>::Graph(void)
{
  for (int i=0; i<MaxGraphSize; i++)
    for (int j=0; j<MaxGraphSize; j++)
      edge[i][j] = 0;
  graphsize = 0;
}
```

Подсчет компонентов графа. Переменная graphsize хранит размер списка вершин. Обращение к этому закрытому члену класса осуществляется посредством метода NumberOfVertices. Оператор GraphEmpty проверяет, пуст ли список.

Доступ к компонентам графа. Компоненты графа содержатся в списке вершин и матрице смежности. Итератор вершин, являясь дружественным по отношению к классу Graph, позволяет сканировать список вершин. Этот итератор — наследник класса SeqListIterator.

```
template ass T>
class VertexIterator: public SeqListIterator<T>
{
  public:
    VertexIterator(Graph<T>& G);
};
```

Конструктор просто инициализирует базовый класс для прохождения списка вершин vertexList.

```
template ass T>
VertexIterator<T>::VertexIterator(Graph<T>& G):
    SeqListIterator<T> (G.vertexList)
{}
```

Итератор сканирует элементы vertexList и используется для реализации функции GetVertexPos, которая осуществляет сканирование списка вершин и возвращает позицию вершины в этом списке.

```
template ass T>
int Graph<T>::GetVertexPos(const T& vertex)
{
  SeqListIterator<T> liter(vertexList);
  int pos = 0;

  while(!liter.EndOfList() && liter.Data() != vertex)
  {
    pos++;
    liter.Next();
  }
  return pos;
}
```

Метод GetWeight возвращает вес ребра, соединяющего vertex1 и vertex2. Чтобы получить позиции этих двух вершин в списке, а следовательно, и строку со столбцом в матрице смежности, используется функция GetVertexPos. Если любая из двух вершин отсутствует в списке вершин, метод возвращает −1.

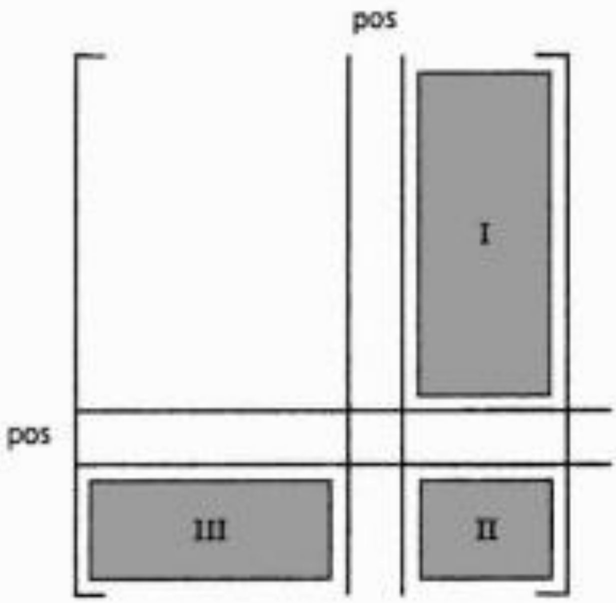
Метод GetNeighbors создает список вершин, смежных с vertex. Этот список является выходным параметром и может быть просканирован с помощью итератора последовательных списков. Если vertex не имеет смежных вершин, метод возвращает пустой список.

```
// возвратить список смежных вершин
template <class T>
SeqList<T>& Graph::GetNeighbors(const T& vertex)
{
  SeqList<T> *L;
  SeqListIterator<T> viter(vertexList);

  // создать пустой список
  L = new SeqList<T>;

  // позиция в списке, соответствующая номеру строки матрицы смежности
  int pos = GetVertexPos(vertex);
  // если вершины vertex нет в списке вершин, закончить
  if (pos == -1)
  {
    cerr << "GetNeighbors: такой вершины нет в графе." << endl;
```

```
    return *L;  // вернуть пустой список
  }
  // сканировать строку матрицы смежности и включать в список
  // все вершины, имеющие ребро ненулевого веса из vertex
  for (int i=0; i<graphsize; i++)
  {
    if (edge[pos][i] > 0)
      L->Insert(viter.Data());
    viter.Next();
  }
  return *L;
}
```

Обновление вершин и ребер. Чтобы вставить ребро, мы используем GetVertexPos для проверки наличия vertex1 и vertex2 в списке вершин. Если какая-либо из них не будет обнаружена, выдается сообщение об ошибке и осуществляется возврат управления. Если позиции pos1 и pos2 получены, метод InsertEdge записывает вес ребра в элемент (pos1, pos2) матрицы смежности. Эта операция выполняется за время $O(n)$, поскольку каждый вызов GetVertexPos требует $O(n)$ времени.

Метод DeleteVertex класса Graph удаляет вершину из графа. Если вершины нет в списке, выдается сообщение об ошибке и осуществляется возврат управления. В противном случае удаляются все ребра, соединяющие удаляемую вершину с другими вершинами. При этом в матрице смежности должны быть скорректированы три области. Поэтому эта операция выполняется за время $O(n^2)$, так как каждая область является частью матрицы $n \times n$.



Область I: Сдвинуть индекс столбца влево
Область II: Сдвинуть индекс строки вверх и индекс столбца влево
Область III: Сдвинуть индекс строки вверх

```
// удалить вершину из списка вершин и скорректировать матрицу
// смежности, удалив принадлежащие этой вершине ребра
template <class T>
void Graph<T>::DeleteVertex(const T& vertex)
{
  // получить позицию вершины в списке вершин
  int pos = GetVertexPos(vertex);
  int row, col;

  // если такой вершины нет, сообщить об этом и вернуть управление
  if (pos == -1)
```

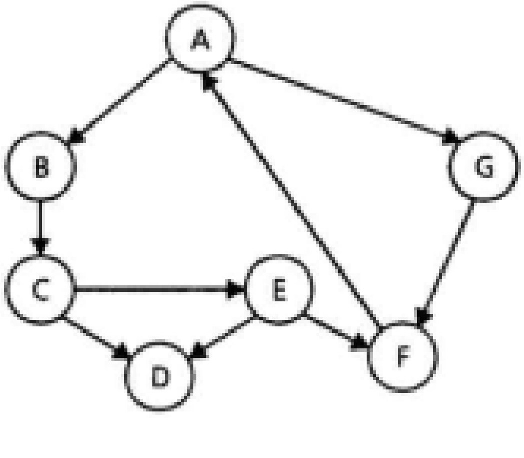
```
  {
    cerr << "DeleteVertex: вершины нет графы" << endl;
    return;
  }

  // удалить вершину и уменьшить graphsize
  vertexList.Delete(vertex);
  graphsize--;

  // матрица смежности делится на три области
  for (row=0; row<pos; row++)              // область I
    for (col=pos+1; col<graphsize; col++)
      edge[row][col-1] = edge[row][col];

  for (row=pos+1; row<graphsize; row++)    // область II
    for (col=pos+1; col<graphsize; col++)
      edge[row-1][col-1] = edge[row][col];

  for (row=pos+1; row<graphsize; row++)    // область III
    for (col=0; col<pos; col++)
      edge[row-1][col] = edge[row][col];
}
```

Удаление ребра производится путем удаления связи между двумя вершинами. После проверки наличия вершин в vertexList метод DeleteEdge присваивает данному ребру нулевой вес, оставляя все другие ребра неизменными. Если такого ребра нет в графе, процедура выдает сообщение об ошибке и завершается.

Способы прохождения графов

Для прохождения нелинейных структур требуется разработать стратегию доступа к узлам и маркирования узлов после обработки. Поисковые методы для бинарных деревьев имеют свои аналоги для графов. В нисходящем обходе бинарного дерева применяется такая стратегия, при которой сначала выполняется обработка узла, а затем уже продвижение вниз по поддереву. Обобщением прямого метода прохождения для графов является поиск **"сначала в глубину" (depth-first)**. Начальная вершина передается в качестве параметра и становится первой обрабатываемой вершиной. По мере продвижения вниз до тупика смежные вершины запоминаются в стеке, с тем чтобы можно было к ним вернуться и продолжить поиск по другому пути в случае, если еще остались необработанные вершины. Обработанные вершины образуют множество всех вершин, достижимых из начальной вершины.

Характерное для деревьев поперечное сканирование начинается с корня, а обход узлов осуществляется уровень за уровнем сверху вниз. Аналогичная стратегия применяется при поиске **"сначала в ширину" (breadth-first)** на гра-

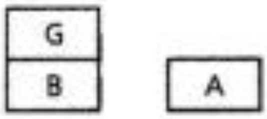


фах, когда, начиная с некоторой начальной вершины, производится обработка каждой смежной с ней вершины. Затем сканирование продолжается на следующем уровне смежных вершин и т.д. до конца пути. При этом для запоминания смежных вершин используется очередь. Проиллюстрируем оба поисковых алгоритма на следующем графе. В данном случае начальной вершиной является A.

Поиск "сначала в глубину". Для хранения обработанных вершин используется список L, а для запоминания смежных вершин — стек S. Поместив начальную вершину в стек, мы начинаем итерационный процесс выталкивания вершины из стека и ее обработки. Когда стек становится пустым, процесс завершается и возвращает список обработанных вершин. На каждом шаге используется следующая стратегия.

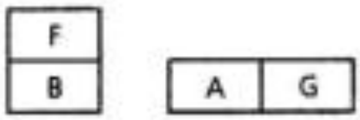
> Вытолкнуть вершину V из стека и проверить по списку L, была ли она обработана. Если нет, произвести обработку этой вершины, а также воспользоваться удобным случаем и получить список смежных с ней вершин. Включить V в список L, чтобы избежать повторной обработки. Поместить в стек те смежные с V вершины, которых еще нет в списке L.

В нашем примере предполагалось, что вершина A является начальной. Поиск начинается с выталкивания A из стека и обработки этой вершины. Затем A включается в список обработанных вершин, а смежные с ней вершины B и G помещаются в стек. После обработки A стек S и список L выглядят следующим образом:



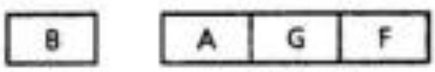
Стек S Список L

Итерация продолжается. Вершина G выталкивается из стека. Так как этой вершины пока нет в списке L, она включается в него, а в стек помещается единственная смежная с ней вершина F:



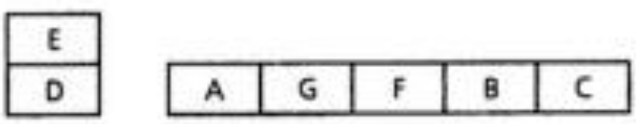
Стек S Список L

Вытолкнув из стека вершину F и поместив ее в список L, мы достигаем тупика, поскольку смежная с F вершина A уже находится в L. В стеке остается вершина B, которая была идентифицирована как смежная с A на первой фазе поиска:



Стек S Список L

Вершины B и C обрабатываются в указанном порядке. Теперь стек содержит вершины D и E, являющиеся смежными с вершиной C:



Стек S Список L

У вершины E две смежные вершины: D и F, и обе они являются подходящими для записи в стек. Однако F уже была обработана на пути A–G–F и поэтому пропускается. Зато вершина D помещается в стек дважды, поскольку наш алгоритм не "знает", что D достижима из C:

D
D

Стек S

A	G	F	B	C	E

Список L

Поиск завершается после обработки вершины D. Второй экземпляр D в стеке игнорируется, так как эта вершина уже находится в списке L:

A	G	F	B	C	E	D

Список L

```
// начиная с начальной вершины, сформировать список вершин,
// обрабатываемых в порядке обхода "сначала в глубину"
template <class T>
SeqList <T> & Graph<T>::DepthFirstSearch(const T& beginVertex)
{
  // стек для временного хранения вершин, ожидающих обработки
  Stack<T> S;

  // L - список пройденных вершин. adjL содержит вершины,
  // смежные с текущей. L создается динамически, поэтому можно
  // возвратить его адрес
  SeqList<T> *L, adjL;
  // iteradjL - итератор списка смежных вершин
  SeqListIterator<T> iteradjL(adjL);
  T vertex;

  // инициализировать выходной список.
  // поместить начальную вершину в стек
  L = new SeqList<T>;
  S.Push(beginVertex);

  // продолжать сканирование, пока не опустеет стек
  while (!S.StackEmpty())
  {
    // вытолкнуть очередную вершину
    vertex = S.Pop();
    // проверить ее наличие в списке L
    if (!FindVertex(*L, vertex))
    {
      // если нет, включить вершину в L,
      // а также получить все смежные с ней вершины
      (*L).Insert(vertex);
      adjL = GetNeighbors(vertex);

      // установить итератор на текущий adjL
      iteradjL.SetList(adjL);

      // сканировать список смежных вершин.
      // помещать в стек те из них, которые отсутствуют в списке L
      for (iteradjL.Reset(); !iteradjL.EndOfList(); iteradjL.Next())
        if (!FindVertex(*L, iteradjL.Data()))
          S.Push(iteradjL.Data());
    }
  }
  // возвратить выходной список
  return *L;
}
```

Поиск "сначала в ширину". Как и в поперечном прохождении бинарного дерева, при поиске "сначала в ширину" для хранения вершин используется очередь, а не стек. Итерационный процесс продолжается до тех пор, пока очередь не опустеет.

> Удалить вершину V из очереди и проверить ее наличие в списке обработанных вершин. Если вершины V нет в списке L, включить ее в этот список. Одновременно получить все смежные с V вершины и вставить в очередь те из них, которые отсутствуют в списке обработанных вершин.

Если применить этот алгоритм к рассмотренному в предыдущем примере графу, то вершины будут обрабатываться в следующем порядке:

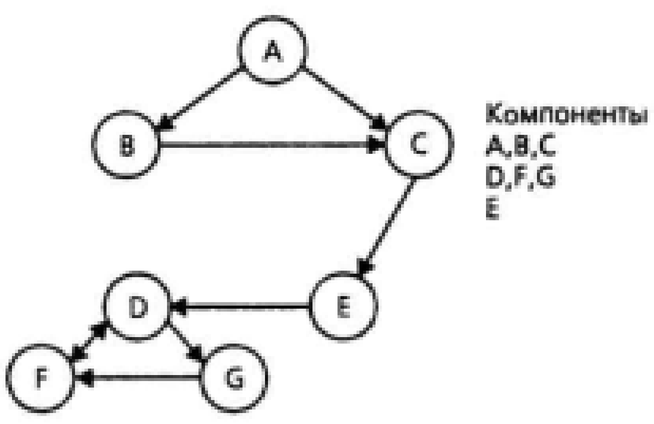
A B G C F D E

Анализ сложности. В описанных алгоритмах поиска посещение каждой вершины требует времени вычислений $O(n)$. При добавлении вершины в список обработанных вершин для обнаружения смежных с ней вершин проверяется строка матрицы смежности. Каждая строка — это $O(n)$, следовательно общее время вычислений равно $n*O(n) = O(n^2)$. Число сравнений, требующихся в случае матричного представлении графа, не зависит от количества ребер в графе. Даже если в графе относительно мало ребер ("разреженный граф"), мы обязаны произвести n сравнений для каждой вершины. В списковом представлении графа быстродействие алгоритма поиска зависит от плотности ребер в графе. В лучшем случае ребер нет и длина каждого списка смежных вершин равна 1. Тогда время вычислений для каждого поиска будет $O(n+n) = O(n)$. В худшем случае каждая вершина связана с каждой и длина каждого списка смежных вершин равна n. Тогда алгоритм поиска имеет порядок $O(n^2)$.

Приложения

Вспомним, что орграф является сильно связанным, если существует направленный путь от любой его вершины к любой другой. **Сильная компонента (strong component)** есть подмножество вершин, сильно связанных друг с другом. Сильно связанный граф имеет одну сильную компоненту, но всякий граф может быть разбит на ряд сильных компонент. Например, на рис. 13.8 граф разбит на три сильных компоненты.

В теории графов для определения сильных компонент используются классические алгоритмы. В данном приложении мы используем для этого поиск "сначала в глубину". Функция PathConnect проверяет существование направленного пути от вершины v к вершине w и возвращает булевские TRUE и FALSE, соответственно.



```
template <class T>
int PathConnect (Graph<T> &G, T v, T w)
{
  SeqList<T> L;

  // найти вершины, связанные с v
  L = G.DepthFirstSearch(v);
  // если w в их числе, вернуть TRUE
  if (L.Find(w))
    return 1;
  else
    return 0;
}
```

Функция ConnectedComponent начинает работу с пустого списка вершин с именем markedList. Этот список все время содержит коллекцию вершин, которые были обнаружены в сильной компоненте. Итератор осуществляет обход вершин графа. Каждая вершина V проверяется на наличие в списке markedList. Если ее там нет, то должна быть построена новая сильная компонента, содержащая V. Список scList, который будет содержать новую сильную компоненту, очищается, и с помощью поиска "сначала в глубину" формируется список L всех вершин, достижимых из V. Для каждой вершины списка L с помощью функции PathConnect проверяется существование пути обратно к V. Если таковой существует, вершина включается в scList и в markedList. Обратите внимание, что вершина вставляется в оба этих списка. Поскольку существует путь от V к каждой вершине из scList и путь от каждой вершины из scList обратно к V, то, следовательно, существует путь между любыми двумя вершинами из scList. Эти вершины и есть очередная сильная компонента. Так как каждая вершина в scList присутствует также в markedList, она не будет рассматриваться повторно.

```
template <class T>
void ConnectedComponent (Graph<T> &G)
{
  VertexIterator<T> viter(G);
  SeqList<T> markedList, scList, L, K;

  for (viter.Reset(); !viter.EndOfList(); viter.Next())
  {
    // проверять в цикле каждую вершину viter.Data()
    if (!markedList.Find(viter.Data()))
    // если не помечен, включить в сильную компоненту
    {
      scList.ClearList();
      // получить вершины, достижимые из viter.Data()
      L = G.DepthFirstSearch(viter.Data());
      // искать в списке вершины, из которых достижима вершина viter.Data()
      SeqListIterator<T> liter(L);
      for (liter.Reset(); !liter.EndOfList(), liter.Next())
        if (PathConnect(G, liter.Data(), viter.Data()))
        {
          // вставить вершины в текущую сильную компоненту и в markedList
          scList.Insert(liter.Data());
          markedList.Insert(liter.Data());
        }
      PrintList(scList); // распечатать сильную компоненту
      cout << endl;
    }
  }
}
```

Программа 13.5. Сильные компоненты

Эта программа находит сильные компоненты в графе, изображенном на рис. 13.8. Граф вводится из файла sctest.dat с помощью ReadGraph. Функции PathConnect, ConnectedComponent и PrintList находятся в файле conncomp.h.

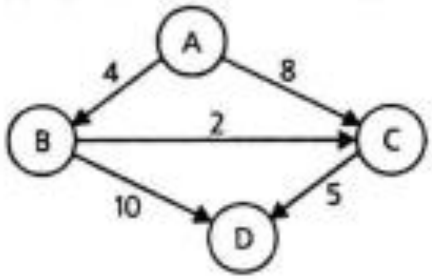
```
#include <iostream.h>
#include <fstream.h>

#include "graph.h"
#include "conncomp.h"

void main(void)
{
  Graph<char> G;

  G.ReadGraph("sctest.dat");

  cout << "Сильные компоненты:" << endl;
  ConnectedComponent(G);
}
/*
<Прогон программы 13.5>

Сильные компоненты:
A B C
D G F
E
*/
```

Минимальный путь. Методы прохождения "сначала в глубину" и "сначала в ширину" находят вершины, достижимые из начальной вершины. При этом движение от вершины к вершине не оптимизируется в смысле минимального пути. Между тем во многих приложениях требуется выбрать путь с минимальной "стоимостью", складывающейся из весов ребер, составляющих путь. Для решения этой задачи мы представляем класс PathInfo. Объект, порождаемый этим классом, описывает путь, существующий между двумя вершинами, и его стоимость. Объекты типа PathInfo запоминаются в очереди приоритетов, которая обеспечивает прямой доступ к объекту с минимальной стоимостью.

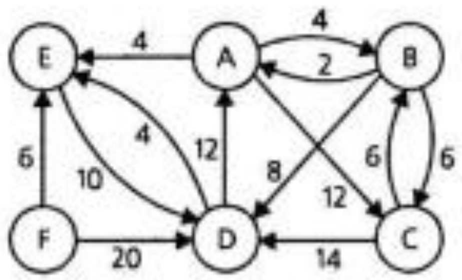
```
template <class T>
struct PathInfo
{
  T startV, endV;
};

template <class T>
int operator <= (const PathInfo<T>& a, const PathInfo<T>& b)
{
  return a.cost <= b.cost;
}
```

Так как между вершинами графа может существовать несколько разных путей, объекты типа PathInfo могут соответствовать одним и тем же вершинам, но иметь разные стоимости. Например, в показанном ниже графе между вершинами A и D существуют три пути с различными стоимостями.

Путь	Стоимость
A - C - D	13
A - B - D	14
A - B - C - D	11



Для сравнения стоимостей в классе PathInfo определен оператор "<=". Алгоритм проверяет объекты типа PathInfo, хранящиеся в очереди приоритетов, и выбирает объект с минимальной стоимостью. Определение минимального пути между начальной (sVertex) и конечной (eVertex) вершинами иллюстрируется на следующем графе. Если между ними нет вообще никакого пути, алгоритм завершается выдачей соответствующего сообщения. Пусть вершина A будет начальной, а D — конечной.



Начать с создания первого объекта типа PathInfo, соединяющего начальную вершину саму с собой при нулевой начальной стоимости. Включить объект в очередь приоритетов.

A — A	0

Очередь приоритетов

Для нахождения минимального пути мы следуем итерационному процессу, который удаляет объекты из очереди приоритетов. Если конечная вершина в объекте есть eVertex, то мы имеем минимальный путь, стоимость которого находится в поле cost. В противном случае просматриваем все вершины, смежные с текущей конечной вершиной endV, и в искомый путь, начинающийся из sVertex, включается еще одно ребро.

В нашем примере мы хотим найти минимальный путь от A до D. Удаляем единственный объект PathInfo, в котором endV = A. Если бы вершина A являлась заданной конечной вершиной eVertex, процесс завершился бы с нулевой минимальной стоимостью. Поскольку вершина A не есть eVertex, она запоминается в списке L, содержащим все вершины, до которых минимальный путь из A известен. Смежными с A вершинами являются вершины B, C и E. Для каждой из них создается объект типа PathInfo, и все эти объекты помещаются в очередь приоритетов. Стоимость пути из A до каждой из этих вершин равна

```
стоимость(A, endV) + вес(endV, <смежная вершина>)
```

Объект PathInfo	startV	endV	Стоимость
$O_{A,B}$	A	B	4
$O_{A,C}$	A	C	12
$O_{A,E}$	A	E	4

Объекты включаются в очередь приоритетов в следующем порядке:

A – B	4	A – E	4	A – C	12

Очередь приоритетов

На следующем шаге объект $O_{A,B}$ удаляется из очереди приоритетов. В нем вершина B есть endV с минимальной стоимостью 4. Поскольку вершины B нет в списке L, она включается в него. Ясно, что не существует последующего пути от A к B со стоимостью меньшей, чем 4. Если бы существовал путь A–X–...–B и смежная с A вершина X находилась бы от нее на расстоянии меньшем, чем 4, то вершина X оказалась бы первой в очереди приоритетов и была бы удалена оттуда раньше вершины B.

Смежными с B вершинами являются A, C и D. Так как A уже в списке L, объекты типа PathInfo создаются для вершин C и D и включаются в очередь приоритетов.

Объект PathInfo	startV	endV	Стоимость
$O_{B,C}$	B	C	10=4+6
$O_{B,D}$	B	C	12=4+8

Результирующая очередь приоритетов содержит четыре элемента. Обратите внимание, что два разных объекта заканчиваются в вершине C. Минимальный из них, имеющий стоимость 10, был только что добавлен в очередь приоритетов и представляет путь A–B–C. Прямой путь A–C, определенный на первом шаге, имеет стоимость 12.

A – E	4	A – C	12	B – C	10	B – D	12

Очередь приоритетов

После удаления объекта $O_{A,E}$ и установления минимальной стоимости пути от A к E равной 4 создается объект $O_{E,D}$ стоимостью 14.

B – C	10	A – C	12	B – D	12	E – D	14

Очередь приоритетов

После очередного удаления объекта с минимальной стоимостью, которым являлся $O_{B,C}$, вершина C может быть добавлена в список L, так как 10 является минимальной стоимостью пути от A к C.

Поскольку конечной вершиной искомого минимального пути является D, мы ожидаем удаления объекта с endV=D. У вершины C есть смежные вершины B и D. Так как B уже обработана, в очередь приоритетов включается только объект $O_{C,D}$.

Объект PathInfo	startV	endV	Стоимость
$O_{C,D}$	C	D	24=10+14

После удаления объекта $O_{A,C}$ из очереди приоритетов он отбрасывается, так как C уже есть в списке. Теперь очередь приоритетов имеет три элемента.

B — D	12	E — D	14	C — D	24

Очередь приоритетов

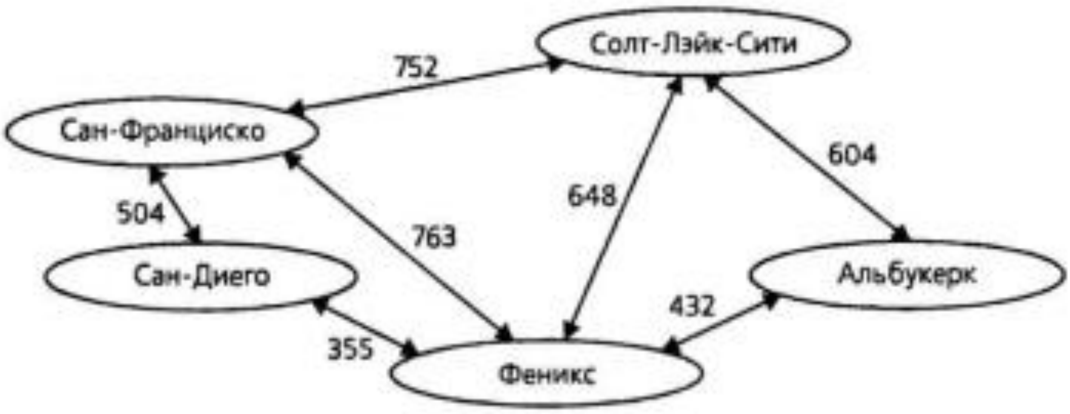
Удаляя $O_{B,D}$ из очереди приоритетов, мы тем самым устанавливаем минимальную стоимость пути от A к D равной 12.

```
template <class T>
int Graph<T>::MinimumPath(const T& sVertex, const T& eVertex)
{
  // очередь приоритетов, в которую помещаются объекты,
  // несущие информацию о стоимости путей из sVertex
  PQueue< PathInfo<T> > PQ(MaxGraphSize);
  // используется при вставке/удалении объектов PathInfo
  // в очереди приоритетов
  PathInfo<T> pathData;
  // L — список всех вершин, достижимых из sVertex и стоимость
  // которых уже учтена. adjL — список вершин, смежных с посещаемой
  // в данный момент. для сканирования adjL используется итератор
  // adjLiter
  SeqList<T> L, adjL;
  SeqListIterator<T> adjLiter(adjL);
  T sv, ev;
  int mincost;

  // сформировать начальный элемент очереди приоритетов
  pathData.startV = sVertex;
  pathData.endV = sVertex;
  // стоимость пути из sVertex в sVertex равна 0
  pathData.cost = 0;
  PQ.PQInsert(pathData);

  // обрабатывать вершины, пока не будет найден минимальный путь
  // к eVertex или пока не опустеет очередь приоритетов
  while (!PQ.PQEmpty())
  {
    // удалить элемент приоритетной очереди и запомнить
    // его конечную вершину и стоимость пути от sVertex
    pathData = PQ.PQDelete();
    ev = pathData.endV;
    mincost = pathData.cost;

    // если это eVertex,
    // то минимальный путь от sVertex к eVertex найден
    if (ev == eVertex)
      break;

    // если конечная вершина уже имеется в L,
    // не рассматривать ее снова
    if (!FindVertex(L, ev))
    {
      // Включить ev в список L
      L.Insert(ev);
      // найти все смежные с ev вершины. Для тех из них,
      // которых нет в L, сформировать объекты PathInfo с начальными
      // вершинами, равными ev, и включить их в очередь приоритетов
      sv = ev;
      adjL = GetNeighbors(sv);
```

```
    // новый список adjL сканируется итератором adjLiter
    adjLiter.SetList(adjL);
    for (adjLiter.Reset(); !adjLiter.EndOfList(); adjLiter.Next())
    {
      ev = adjLiter.Data();
      if (!FindVertex(L,ev))
      {
        // создать новый элемент приоритетной очереди
        pathData.startV = sv;
        pathData.endV = ev;
        // стоимость равна текущей минимальной стоимости
        // плюс стоимость перехода от sv к ev
        pathData.cost = mincost + GetWeight(sv, ev);
        PQ.PQInsert(pathData);
      }
    }
  }
}

if (ev == eVertex)
  return mincost;
else
  return -1;
}
```

Программа 13.6. Система авиаперевозок

Транспортная система авиакомпании имеет список городов на некотором маршруте полетов. Пользователь вводит исходный город, а процедура определения минимального пути выдает кратчайшие расстояния между этим городом и всеми прочими пунктами назначения. Эта авиалиния соединяет главные города на Западе.




```
#include <iostream.h>
#include <fstream.h>

#include "strclass.h"
#include "graph.h"        // метод MinimumPath

void main(void)
{
  // вершины типа символьных строк (названия городов)
  Graph<String> G;
  String S;
```

```
  // ввод описания транспортного графа
  G.ReadGraph("airline.dat");

  // запросить аэропорт отправления
  cout << "Выдать мин. расстояние при отправлении из ";
  cin >> S;

  // с помощью итератора пройти список городов и определить
  // мин. расстояния от точки отправления
  VertexIterator<String> viter(G);
  for (viter.Reset(); !viter.EndOfList(); viter.Next())
    cout << "Минимальное расстояние от аэропорта " << S <<
         << " до аэропорта " << viter.Data() << " = "
         << G.MinimumPath(S, viter.Data()) << endl;
}
/*

<Прогон #1 программы 13.6>

Выдать минимальное расстояние при отправлении из Солт-Лэйк-Сити
Мин. расстояние от аэропорта Солт-Лэйк-Сити до аэропорта Солт-Лэйк-Сити = 0
Мин. расстояние от аэропорта Солт-Лэйк-Сити до аэропорта Альбукерк = 604
Мин. расстояние от аэропорта Солт-Лэйк-Сити до аэропорта Феникс = 648
Мин. расстояние от аэропорта Солт-Лэйк-Сити до аэропорта Сан-Франциско = 752
Мин. расстояние от аэропорта Солт-Лэйк-Сити до аэропорта Сан-Диего = 1003

<Прогон #2 программы 13.6>

Выдать мин. расстояние при отправлении из Сан-Франциско
Мин. расстояние от аэропорта Сан-Франциско до аэропорта Солт-Лэйк-Сити = 752
Мин. расстояние от аэропорта Сан-Франциско до аэропорта Альбукерк = 1195
Мин. расстояние от аэропорта Сан-Франциско до аэропорта Феникс = 763
Мин. расстояние от аэропорта Сан-Франциско до аэропорта Сан-Франциско = 0
Мин. расстояние от аэропорта Сан-Франциско до аэропорта Сан-Диего = 504
*/
```

Достижимость и алгоритм Уоршалла

Для каждой пары вершин некоторого графа говорят, что V_j достижима из V_i тогда и только тогда, когда существует направленный путь от V_i к V_j. Это определяет **отношение достижимости R (reachability relation R)**. Для каждой вершины V_i поиск "сначала в глубину" находит все вершины, достижимые из V_i. При использовании поиска "сначала в ширину" получается серия списков достижимости, которые образуют отношение R:

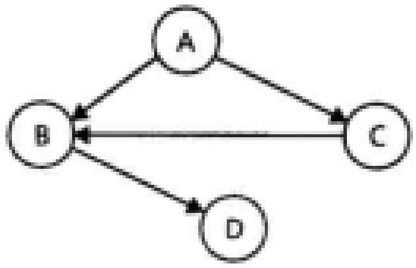
V_1: <Список достижимости для V_1>

V_2: <Список достижимости для V_2>

. . .

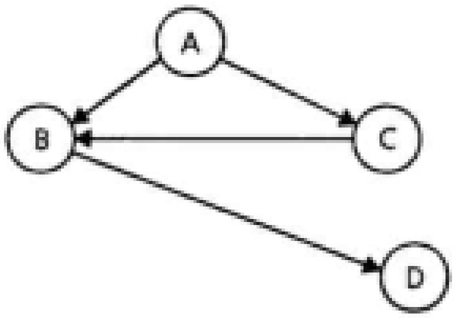
Vn: <Список достижимости для V_n>

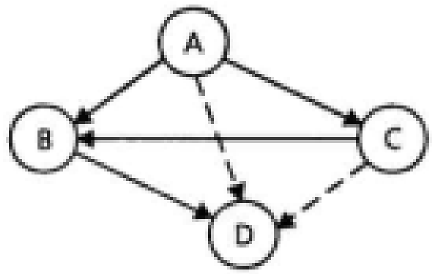
Это же отношение можно также описать с помощью **матрицы достижимости (reachability matrix)** размером n × n, которая содержит 1 в позиции (i,j), представляя тем самым V_i R V_j. В следующем примере показаны списки и матрица достижимости для изображенного здесь графа.



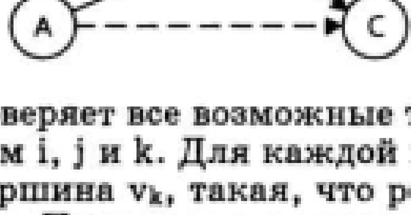
Списки достижимости	Матрица достижимости
A: A B C D	1 1 1 1
B: B D	0 1 0 1
C: C B	0 1 1 0
D:	0 0 0 1

Матрицу достижимости можно использовать для проверки существования пути между двумя вершинами. Если элемент (i,j) равен 1, то существует минимальный путь между V_i и V_j. Вершины в списке достижимости можно использовать для наращивания ребер в исходном графе. Если существует путь из вершины v к вершине w (w достижима из v), мы добавляем ребро E(v,w), соединяющее эти две вершины. Расширенный граф G1 состоит из вершин V графа G и ребер, связывающих вершины, которые соединены направленным путем. Такой расширенный граф называется **транзитивным замыканием** (transitive closure). Ниже приводится пример графа и его транзитивного замыкания.





Задача нахождения списка достижимости с помощью поиска "сначала в глубину" предлагается читателю в качестве упражнения. Более изящный подход применяется в знаменитом алгоритме Стефана Уоршалла. Матрица достижимости некоторого графа может быть построена путем присвоения 1 каждой паре вершин, связанных общей вершиной. Предположим, мы строим матрицу достижимости R и вершинам a, b, c соответствуют индексы i, j, k. Если R[i][j] = 1 и R[i][k] = 1, установить R[i][j] = 1.



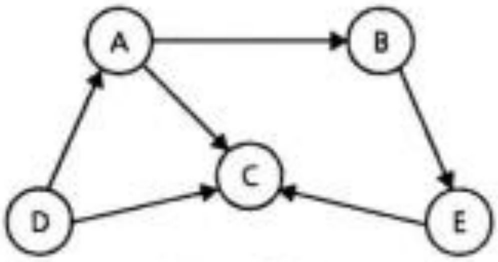
Алгоритм Уоршалла проверяет все возможные тройки с помощью трех вложенных циклов по индексам i, j и k. Для каждой пары (i,j) добавляется ребро $E(v_i,v_j)$, если существует вершина v_k, такая, что ребра $E(v_i,v_k)$ и $E(v_k,v_j)$ находятся в расширенном графе. Повторяя этот процесс, мы соединяем дополнительными ребрами любую пару достижимых вершин. В результате получается матрица достижимости.

Предположим, что вершины v и w достижимы через направленный путь, связывающий пять вершин. Тогда существует последовательность вершин, формирующих путь

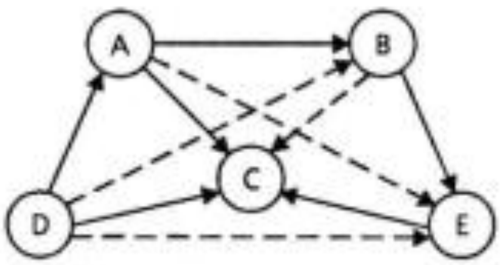
$v = x_1, x_2, x_3, x_4, x_5 = w$

Имея путь от v до w, мы должны показать в матрице достижимости, что алгоритм Уоршалла в конце концов даст тот же путь. С помощью трех вложенных циклов мы проверяем все возможные тройки вершин. Допустим, вершины идут в порядке x_1–x_5. В процессе просмотра различных троек вершина x_2 идентифицируется как общий узел, связывающий x_1 и x_3. Следовательно, согласно алгоритма Уоршалла мы вводим новое ребро $E(x_1,x_3)$. Для пары x_1, x_4 общей связующей вершиной является x_3, так как путь, соединяющий x_1 и x_3, был найден на предыдущем шаге итерации. Поэтому мы создаем ребро $E(x_1,x_4)$. Таким же образом x_4 становится общей вершиной, связывающей x_1 и x_5, и мы добавляем ребро $E(x_1,x_5)$ и присваиваем элементу R[1][5] значение 1.

Проиллюстрируем алгоритм Уоршалла на следующем графе. Здесь дополнительные ребра, добавленные для формирования транзитивного замыкания, изображены пунктиром.



Исходный граф



Транзитивное замыкание

Матрица достижимости

1 1 1 0 1
0 1 1 0 1
0 0 1 0 0
1 1 1 1 1
0 0 1 0 1

Алгоритм Уоршалла имеет время вычисления $O(n^3)$. При сканировании матрицы смежности применяются три вложенных цикла. Списковое представление графа также дает сложность $O(n^3)$.

Программа 13.7. Использование алгоритма Уоршалла

Алгоритм Уоршалла используется для создания и печати матрицы достижимости.

```
#include <iostream.h>
#include <fstream.h>

#include "graph.h"

template <class T>
void Warshall(Graph<T>& G)
{
  VertexIterator<T> vi(G), vj(G);
```

```
    int i, j, k;
    int WSM[MaxGraphSize][MaxGraphSize];  // матрица Уоршалла
    int n = G.NumberOfVertices();

    // создать исходную матрицу
    for (vi.Reset(); i=0; !vi.EndOfList(); vi.Next(), i++)
      for (vj.Reset(); i=0; !vj.EndOfList(); vj.Next(), j++)
        if (i == j)
          WSM[i][i] = 1;
        else
          WSM[i][j] = G.GetWeight(vi.Data(), vj.Data());

    // просмотреть все тройки. записать 1 в WSM, если существует ребро
    // между vi и vj или существует тройка vi-vk-vj, соединяющая
    // vi и vj
    for (i=0; i<n; i++)
      for (j=0; j<n; j++)
        for (k=0; k<n; k++)
          WSM[i][j] |= WSM[i][k] & WSM[k]j];

    // распечатать каждую вершину и ее строку из матрицы достижимости
    for (vi.Reset(); i=0; !vi.EndOfList(); vi.Next(), i++)
    {
      cout << vi.Data() << ": ";
      for (j=0; j<n; j++)
        cout << WSM[i][j] << " ";
      cout << endl;
    }
}
void main(void)
{
    Graph<char> G;

    G.ReadGraph("warshall.dat");

    cout << "Матрица достижимости:" << endl;
    Warshall(G);
}
/*
<Прогон программы 13.7>

Матрица достижимости:
A: 1 1 1 0 1
B: 0 1 1 0 1
C: 0 0 1 0 0
D: 1 1 1 1 1
E: 0 0 1 0 1
*/
```

Источники:

1. У. Топп, У. Форд – Структуры данных в C++
2. https://bookflow.ru/chto-takoe-graf-klassifikatsiya-grafov-realizatsiya-na-c/ - понятие графов, примеры кода
3. https://habr.com/ru/post/582206/ - библиотеки