§1. Определение графа

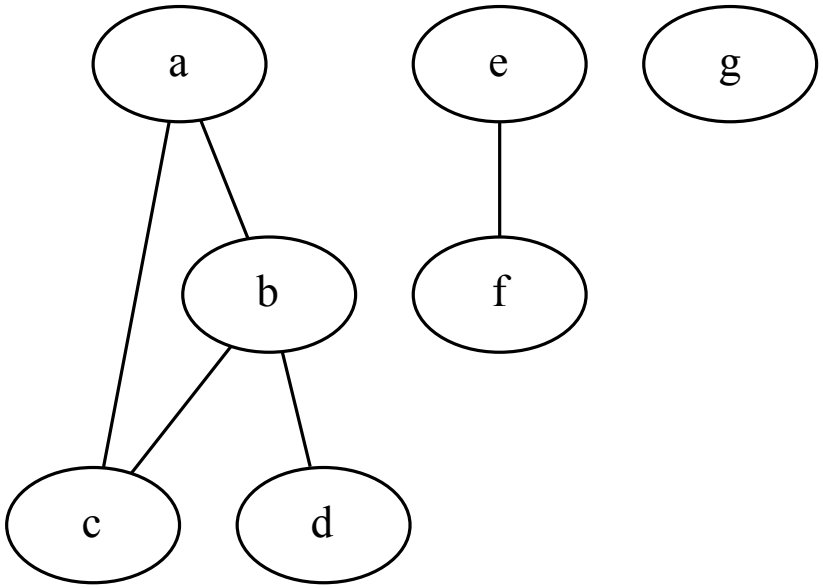
Определение 1.1. Пара $\langle V, E \rangle$ называется *простым графом*, если V – непустое конечное множество элементов, называемых *вершинами*, а E – множество неупорядоченных пар различных элементов из V, называемых *рёбрами*.

Обозначения. Пусть G – граф, тогда $V(G)$ – множество его вершин, а $E(G)$ – множество его рёбер.

Если ребро $\{u, v\}$ принадлежит $E(G)$, то говорят, что оно *соединяет* вершины u и v. При этом вершины u и v называют *смежными*. Также их называют *инцидентными* ребру $\{u, v\}$ (а ребро – *инцидентным* этим вершинам).

Два различных ребра графа называют *смежными*, если они инцидентны общей вершине.

Пример. (Простой граф G.)



$$V(G) = \{a, b, c, d, e, f, g\}$$
$$E(G) = \{\{a, b\}, \{b, c\}, \{a, c\}, \{b, d\}, \{e, f\}\}$$

Граф на языке DOT:

```
1  graph {
2        a
3        b
4        c
5        d
6        e
7        f
8        g
9        a--b
10       b--c
11       b--d
12       a--c
13       e--f
14 }
```

Степень вершины v – это количество рёбер, инцидентных вершине v. Степень обозначается как $\deg v$.

Вершина степени 0 называется *изолированной вершиной*. Вершина степени 1 называется *концевой вершиной*. Ребро, инцидентное концевой вершине, также называют *концевым ребром*.

Утверждение 1.1. Сумма степеней вершин графа G равна удвоенному числу его рёбер: $\sum\limits_{u \in V(G)} \deg u = 2\,|E\,(G)|$.

▷ Доказательство утверждения тривиально: поскольку любое ребро инцидентно двум вершинам, в сумме степеней всех вершин графа каждое ребро учитывается дважды. ◁

В программировании чаще всего применяются не простые графы, а так называемые *мультиграфы*. В мультиграфе две вершины могут соединяться несколькими рёбрами, и допускаются *петли*, то есть рёбра, соединяющие вершину с ней самой.

Мультиграф можно определить через введение понятия *мультимножества* рёбер, разрешающего повторное вхождение одного и того же элемента.

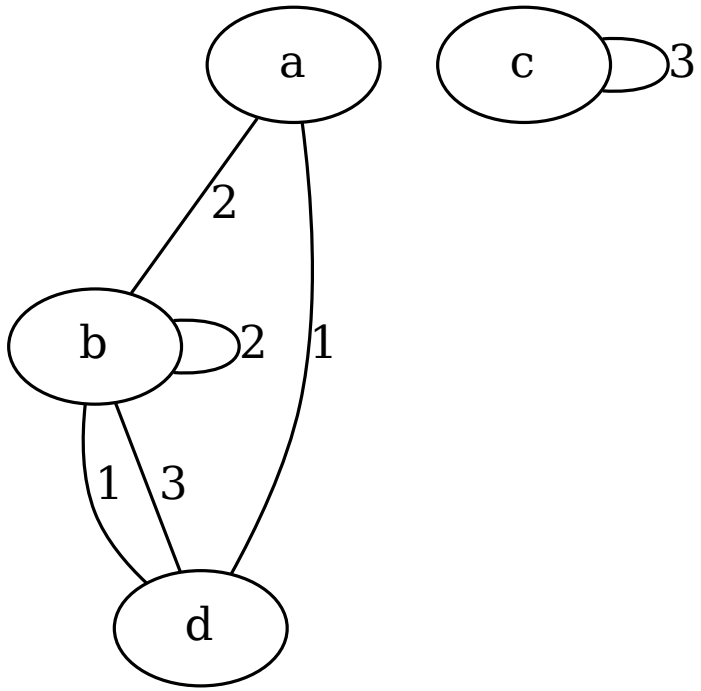
Возможен и другой путь. Принимая во внимание, что на практике соединение двух вершин несколькими рёбрами имеет смысл только в случае, если эти рёбра чем-то отличаются, мы будем помечать рёбра мультиграфа некоторыми *атрибутами*, и мультимножество нам не понадобится.

Определение 1.2. Тройка $\langle V, A, E\rangle$ называется *размеченным мультиграфом*, если V – непустое конечное множество вершин, A – множество атрибутов, а E – отображение множества одно- и двухэлементных подмножеств множества V в A.

Элементами множества E являются упорядоченные пары вида $\langle\{u, v\}, a\rangle$ и $\langle\{u\}, a\rangle$. Их мы будем называть рёбрами мультиграфа. Очевидно, что пары $\langle\{u\}, a\rangle$ соответствуют петлям.

Мы будем считать, что петля добавляет 2 к степени инцидентной ей вершины.

Пример. (Мультиграф G.)



$V(G) = \{a, b, c, d\}$
$A(G) = \{1, 2, 3\}$
$E(G) = \{\langle\{a, b\}, 2\rangle, \ldots, \langle\{c\}, 3\rangle\}$

Мультиграф на языке DOT:

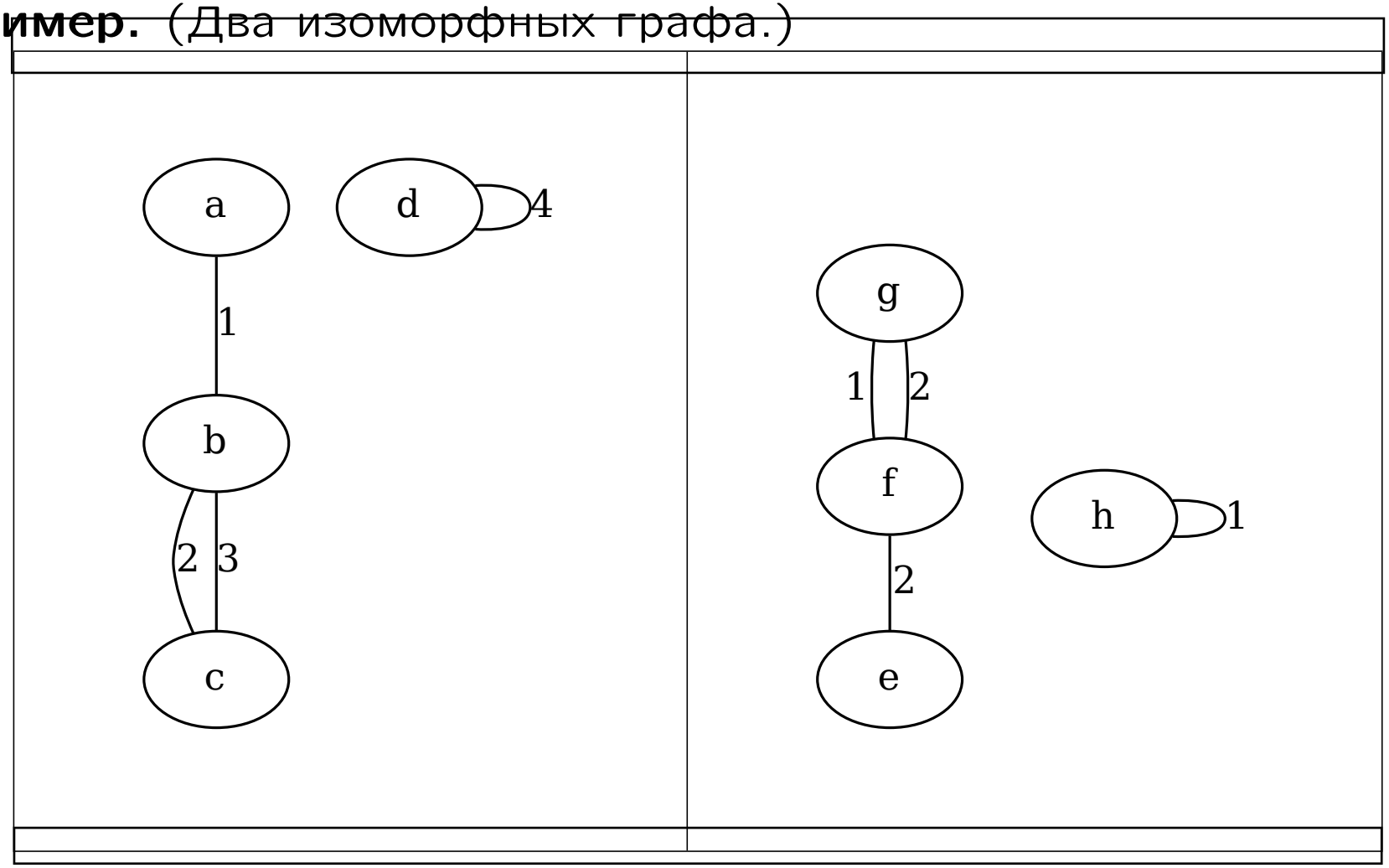
```
1  graph {
2        a
3        b
4        c
5        d
6        a--d [label = "1"]
7        a--b [label = "2"]
8        d--b [label = "3"]
9        b--d [label = "1"]
10       b--b [label = "2"]
11       c--c [label = "3"]
12 }
```

Так как мы не определяем, какими именно объектами являются вершины графа, то de facto можно считать, что в любом графе вершины помечены какой-то информацией. Более того, как мы уже выяснили, рёбра тоже могут быть помечены атрибутами. Понятие *изоморфизма* позволяет абстрагироваться от разметки и учитывать только структуру (форму) графа.

Определение 1.3. Два графа G_1 и G_2 называются *изоморфными*, если существует взаимно однозначное соответствие между множествами их вершин, обладающее тем свойством, что число рёбер, соединяющих любые две вершины в G_1 (необязательно различные), равно числу рёбер, соединяющих соответствующие вершины в G_2.

Обозначение. $G_1 \cong G_2$.

Пример. (Два изоморфных графа.)

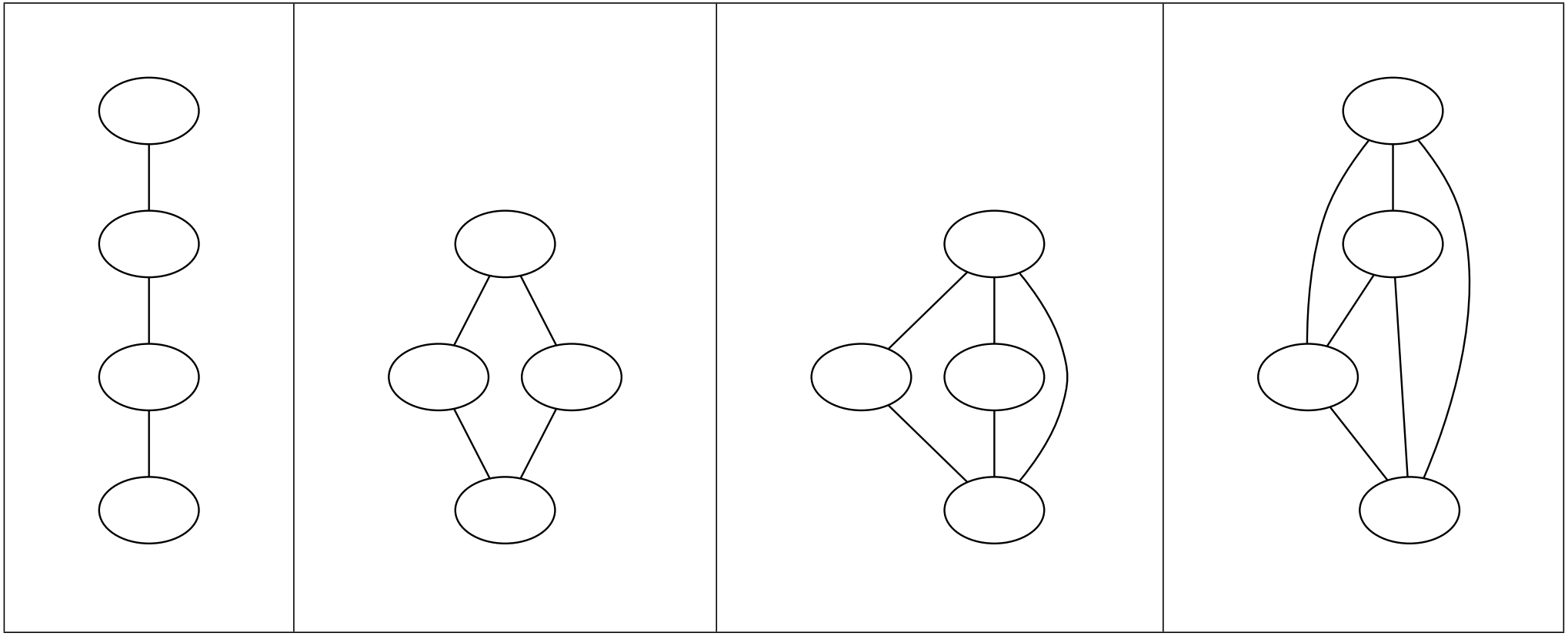


Взаимно однозначное соответствие:

$\langle a, e\rangle, \langle b, f\rangle, \langle c, g\rangle, \langle d, h\rangle$.

Определение 1.4. *Абстрактный граф* – это множество изоморфных помеченных графов.

Пример. (Схематическое изображение некоторых абстрактных графов, имеющих 4 вершины)



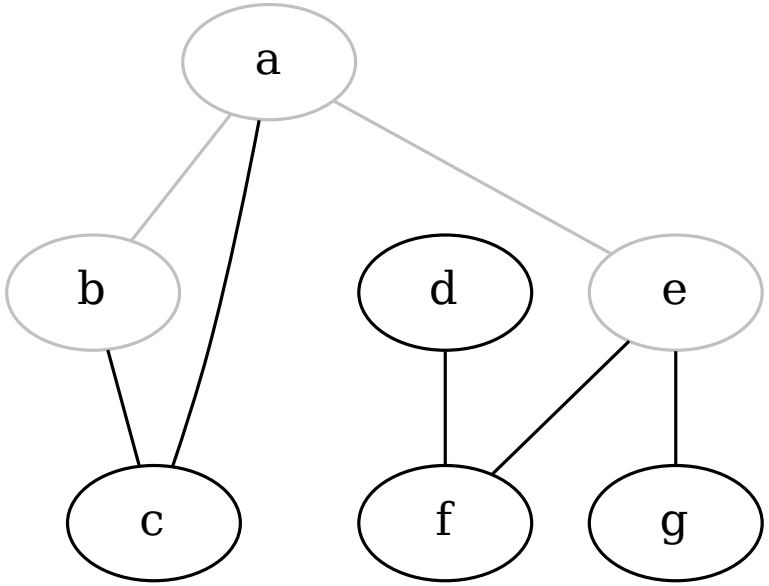
§2. Подграфы

Определение 2.1. Граф $G' = \langle V', E' \rangle$ называют *подграфом* графа $G = \langle V, E \rangle$, если $V' \subseteq V$ и $E' \subseteq E$, причём ребро $\{u, v\}$ может содержаться в E' только тогда, когда $u \in V'$ и $v \in V'$.

В свою очередь, граф G по отношению к своему подграфу G' является *надграфом*.

Определение 2.2. *Остовным* называется подграф $G' = \langle V', E' \rangle$, включающий в себя все вершины надграфа $G = \langle V, E \rangle$: $V = V'$.

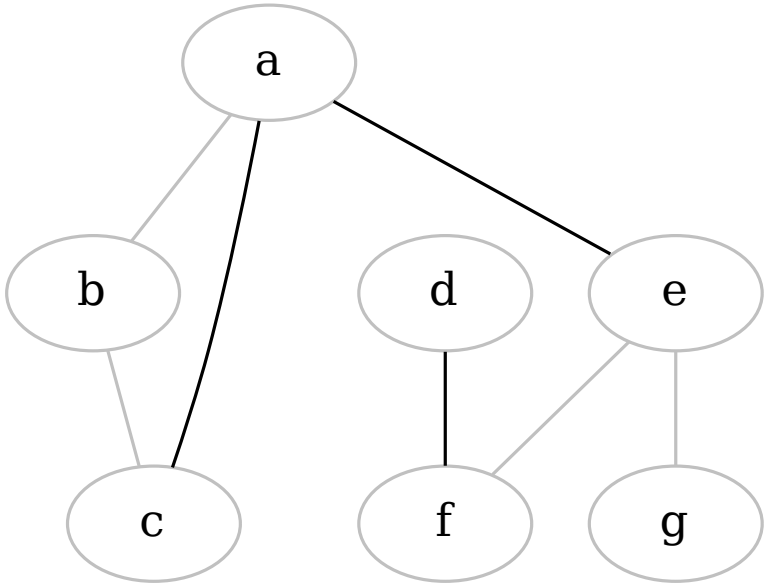
Пример. (Подграф.)



На языке DOT:

```
1  graph {
2        a [color=gray]
3        b [color=gray]
4        c; d
5        e [color=gray]
6        f; g
7        a--b [color=gray]
8        b--c
9        c--a
10       a--e [color=gray]
11       e--g
12       d--f
13       e--f
14 }
```

Пример. (Остовный подграф.)



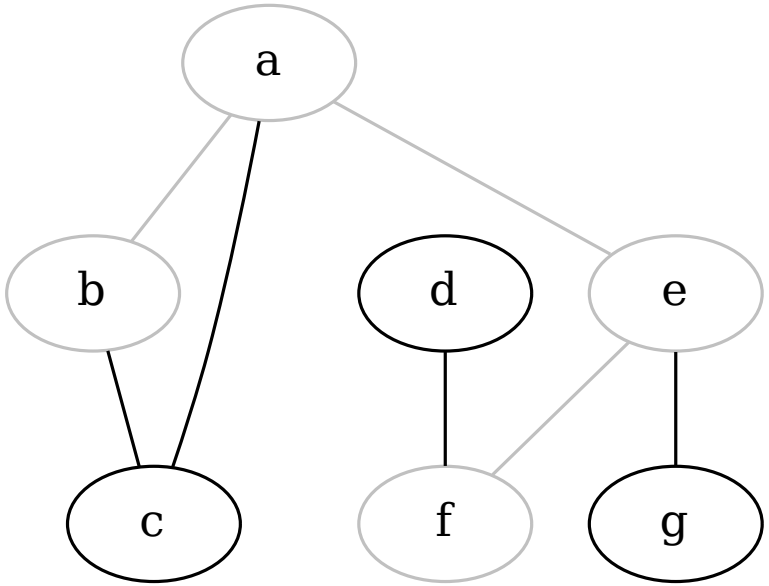
На языке DOT:

```
graph {
      a [color=gray]
      b [color=gray]
      c [color=gray]
      d [color=gray]
      e [color=gray]
      f [color=gray]
      g [color=gray]
      a--b [color=gray]
      b--c [color=gray]
      c--a
      a--e
      e--g [color=gray]
      d--f
      e--f [color=gray]
}
```

Определение 2.3. *Вершинно-порождённым* называется подграф $G' = \langle V', E' \rangle$, включающий в себя те и только те рёбра надграфа $G = \langle V, E \rangle$, концы которых принадлежат V'.

Определение 2.4. *Рёберно-порождённым* называется подграф $G' = \langle V', E' \rangle$, включающий в себя те и только те вершины надграфа $G = \langle V, E \rangle$, которые инцидентны рёбрам из E'.

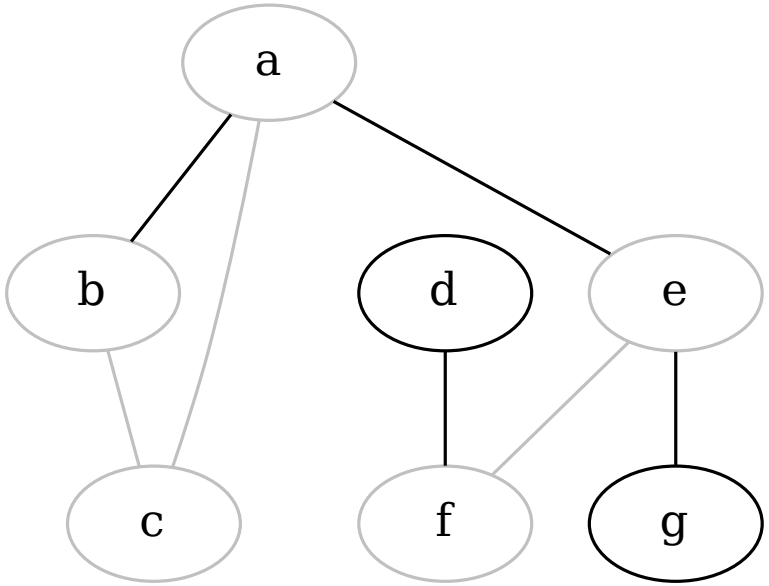
Пример. (Верш.-порожд. подграф.)



На языке DOT:

```
graph {
      a [color=gray]
      b [color=gray]
      c; d
      e [color=gray]
      f [color=gray]
      g
      a--b [color=gray]
      b--c
      c--a
      a--e [color=gray]
      e--g
      d--f
      e--f [color=gray]
}
```

Пример. (Рёб.-порожд. подграф.)



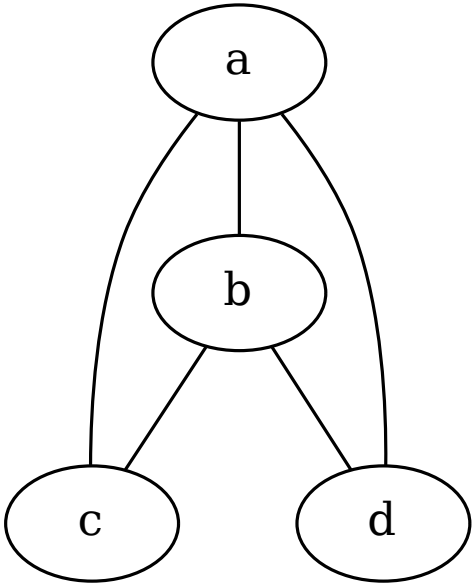
На языке DOT:

```
graph {
      a [color=gray]
      b [color=gray]
      c [color=gray]
      d
      e [color=gray]
      f [color=gray]
      g
      a--b
      b--c [color=gray]
      c--a [color=gray]
      a--e
      e--g
      d--f
      e--f [color=gray]
}
```

§3. Маршруты в графе

Определение 3.1. *Маршрутом* в графе называется конечная последовательность рёбер, в которой любые два последовательных ребра различны и смежны. *Длина маршрута* – это количество принадлежащих ему рёбер. Маршрут нулевой длины называется *тривиальным*.

Пример.



1. $\{a,b\},\{b,c\},\{a,c\},\{a,b\}$ – маршрут;
2. $\{a,b\},\{b,c\},\{b,d\}$ – маршрут;
3. $\{a,b\},\{b,c\},\{b,c\}$ – не маршрут;
4. $\{a,b\},\{b,c\},\{a,d\}$ – не маршрут.

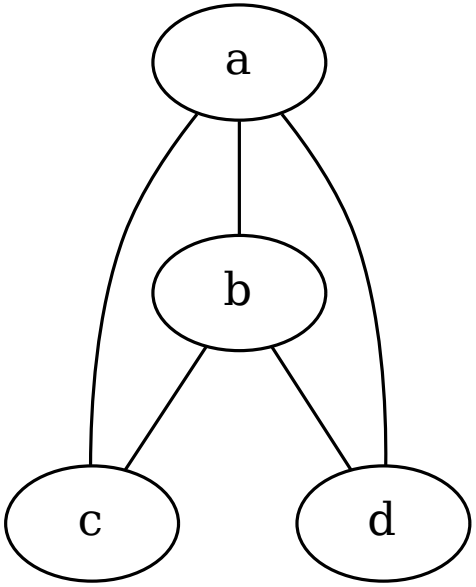
Определение 3.2. Маршрут называется *цепью*, если все принадлежащие ему рёбра различны.

Говорят, что цепь $\{u, x_1\}, \{x_1, x_2\}, \ldots, \{x_{n-1}, x_n\}, \{x_n, v\}$ *соединяет* вершины u и v.

Говорят, что цепь $\{u, v\}$ единичной длины *соединяет* вершины u и v.

Тривиальная цепь *соединяет* любую вершину с ней самой.

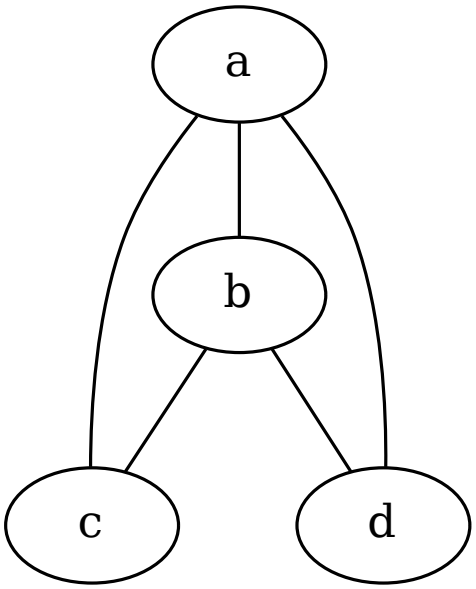
Пример.



1. $\{a, b\}, \{b, c\}, \{a, c\}, \{a, b\}$ – не цепь;
2. $\{a, b\}, \{b, c\}, \{b, d\}$ – цепь, соединяющая a и d;
3. $\{a, b\}, \{b, c\}$ – цепь, соединяющая a и c;
4. $\{a, b\}, \{b, d\}, \{a, d\}$ – цепь, соединяющая a и a;
5. $\{a, c\}$ – цепь, соединяющая a и c.

Определение 3.3. Цепь называется *простой*, если каждая вершина графа инцидентна не более, чем двум принадлежащим этой цепи ребрам.

Пример.



1. $\{a,b\},\{b,c\},\{b,d\}$ – не простая цепь;
2. $\{a,b\},\{b,c\}$ – простая цепь;
3. $\{a,b\},\{b,d\},\{a,d\}$ – простая цепь.
4. $\{a,c\},\{b,c\},\{a,b\},\{a,d\}$– не простая цепь.

Простую цепь, имеющую вид
$\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{m-1}, v_m\}$
принято записывать как
$v_0 \leftrightarrow v_1 \leftrightarrow v_2 \leftrightarrow \ldots \leftrightarrow v_m$.
В этой записи хорошо видно, что цепь соединяет вершины v_0 и v_m.

Определение 3.4. *Циклом* (*простым циклом*) называется цепь (простая цепь), соединяющая вершину с ней самой.

Определения маршрута, (простой) цепи и (простого) цикла справедливы и для мультиграфов.

Простую цепь в мультиграфе, имеющую вид
$\langle \{v_0, v_1\}, a_1 \rangle, \langle \{v_1, v_2\}, a_2 \rangle, \ldots, \langle \{v_{m-1}, v_m\}, a_m \rangle$
можно записывать как
$v_0 \overset{a_1}{\leftrightarrow} v_1 \overset{a_2}{\leftrightarrow} v_2 \overset{a_3}{\leftrightarrow} \ldots \overset{a_m}{\leftrightarrow} v_m$.

Замечание. Из определения следует, что любая вершина графа либо вообще не инцидентна ни одному ребру простого цикла в этом графе, либо инцидентна сразу двум рёбрам.

Утверждение 3.1. Из множества рёбер таких, что любая вершина графа либо вообще не инцидентна ни одному ребру из этого множества, либо инцидентна сразу двум рёбрам, можно составить один или несколько простых циклов.

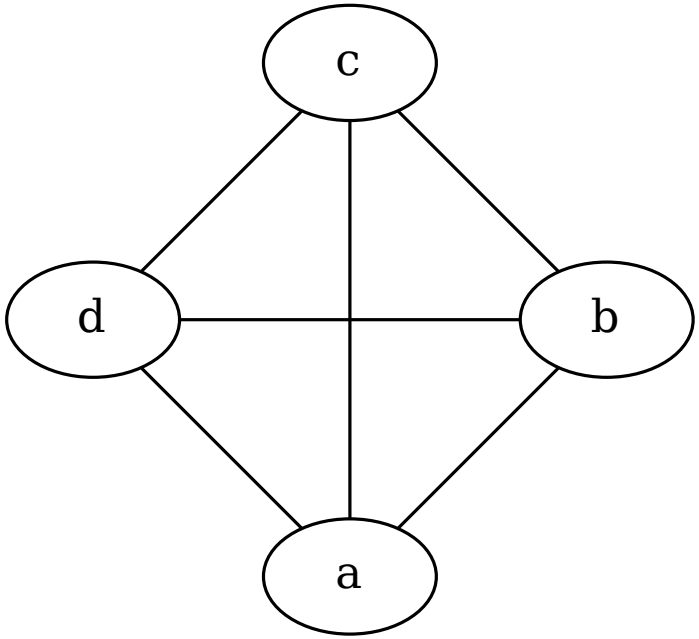
Утверждение 3.2. Если вершины u и v графа соединены двумя различными простыми цепями, то в графе существует один или несколько простых циклов, составленных из рёбер, принадлежащих симметрической разности этих цепей.

§4. Классификация графов

Определение 4.1. Граф называется *пустым*, если он вообще не имеет рёбер.

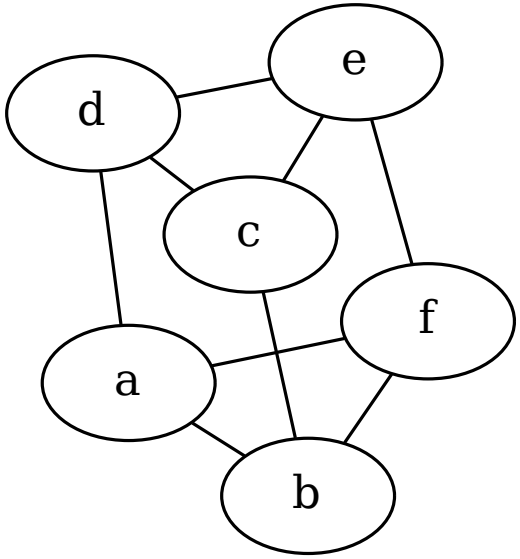
Определение 4.2. Простой граф, имеющий максимальное количество рёбер, называется *полным*.

Пример. (Полный граф.)



Определение 4.3. Граф, у которого все вершины имеют одну и ту же степень, называется *регулярным*.

Пример. (Регулярный граф.)

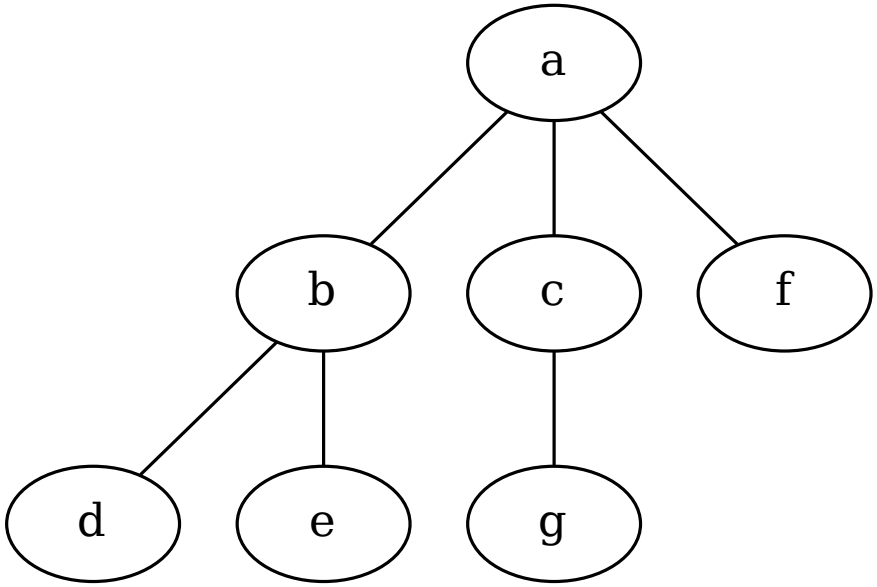


Определение 4.4. Граф *несвязен*, если множество его вершин распадается на два (или более) непересекающихся подмножества, таких, что нет ни одного ребра, концы которого принадлежат разным подмножествам. В противном случае, граф называется *связным*.

Другими словами, несвязный граф состоит из двух и более частей, не соединённых рёбрами. Эти части называются *компонентами связности*.

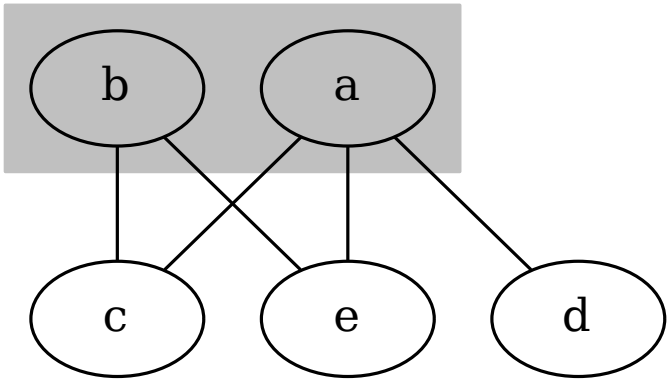
Определение 4.5. *Дерево* – это связный граф с минимальным количеством рёбер: $|E(G)| = |V(G)| - 1$.

Пример. (Дерево.)



Определение 4.6. Граф называется *двудольным*, если существует такое разбиение множества его вершин на два непересекающихся подмножества, что концы любого ребра принадлежат разным подмножествам. (В общем случае определение может быть расширено на k-дольные графы.)

Пример. (Двудольный граф.)



Утверждение 4.1. Граф является двудольным тогда и только тогда, когда он не содержит нетривиальных простых циклов нечётной длины.

▷ Необходимость. Предположим, что некоторый двудольный граф содержит нетривиальный простой цикл длины m, где m – нечётно.

Так как граф – двудольный, то множество вершин цикла разбивается на два непересекающихся подмножества – V_1 и V_2, и концы любого ребра цикла принадлежат разным подмножествам.

Пусть $|V_1| = k$. Каждой вершине из V_1 инцидентно два ребра цикла. При этом не существует ребра, инцидентного сразу двум вершинам из V_1, иначе эти две вершины не могли бы обе принадлежать V_1. Поэтому все рёбра, инцидентные вершинам из V_1, различны, и всего этих рёбер – $2k$.

Один конец любого ребра цикла принадлежит V_1. Так как цикл – простой, то все рёбра в нём различны. Значит количество рёбер, инцидентных вершинам из V_1, равно m. Тем самым $m = 2k$ – чётное.

Достаточность. Пусть граф не содержит нетривиальных простых циклов нечётной длины.

Будем последовательно рассматривать все компоненты связности этого графа и разделять вершины каждого компонента на два непересекающихся подмножества, помеченных знаками «+» и «−», таких, что концы любого ребра компоненты принадлежат разным подмножествам.

Возьмём произвольную вершину v компоненты связности и пометим её знаком «+». Покажем, что любую вершину u той же компоненты мы можем пометить знаком «−», если простая цепь из v в u имеет нечётную длину, и знаком «+» – в противном случае.

Если длины всех простых цепей, соединяющих v и u, имеют одинаковую чётность, то, очевидно, никакой проблемы нет. Поэтому предположим, что существуют две соединяющие v и u простые цепи, одна из которых имеет чётную длину, а другая – нечётную. Эти две цепи образуют нетривиальный цикл $v \leftrightarrow \ldots \leftrightarrow u \leftrightarrow \ldots \leftrightarrow v$ нечётной длины. Если этот цикл – непростой, значит какая-то вершина w входит в него дважды и разбивает цикл на два подцикла, один из которых имеет нечётную длину. Повторяя рассуждение для этого подцикла, мы в конце концов придём к тому, что в графе существует простой цикл нечётной длины, что невозможно.

Теперь покажем, что вершины, имеющие один знак, не могут быть смежными. Для этого предположим, что две смежные вершины u_1 и u_2 имеют один знак. Тогда, опять же, в графе имеется нетривиальный цикл $v \leftrightarrow \ldots \leftrightarrow u_1 \leftrightarrow u_2 \leftrightarrow \ldots \leftrightarrow v$ нечётной длины, что невозможно. $\lhd$

§5. Матрицы графов

Определение 5.1. *Матрица смежности* простого графа G – это симметричная квадратная матрица $A = \left[a_{i,j}\right]$ порядка $n = |V(G)|$, в которой элемент $a_{i,j}$ равен 1, если в графе есть ребро $\left\{v_i, v_j\right\}$, и 0, если такого ребра нет.

Из определения следует, что сумма элементов i-той строки матрицы смежности равна степени вершины v_i, а сумма элементов j-того столбца равна степени вершины v_j.

Матрица смежности пустого графа состоит из одних нулей, а матрица смежности полного графа – из одних единиц.

Определение 5.2. *Матрица смежности* мультиграфа G – это симметричная квадратная матрица $A = \left[a_{i,j}\right]$ порядка $n = |V(G)|$, в которой элемент $a_{i,j}$ равен множеству атрибутов рёбер $\left\{v_i, v_j\right\}$, если они есть в графе, и пустому множеству в противном случае.

Если граф несвязен и имеет q компонент, то путём перестановки строк и столбцов его матрица смежности может быть приведена к блочно-диагональному виду, где каждый диагональный блок $A_{i,i}$ является матрицей смежности соответствующей компоненты.

$$A = \begin{bmatrix} A_{11} & 0 & \cdots & 0 \\ 0 & A_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{qq} \end{bmatrix}$$

(В случае мультиграфа элементы нулевых блоков – пустые множества.)

В случае k-дольного графа его матрица смежности может быть приведена к блочному виду, когда по главной диагонали идут только нулевые блоки:

$$A = \begin{bmatrix} 0 & A_{12} & \cdots & A_{1k} \\ A_{21} & 0 & \cdots & A_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k1} & A_{k2} & \cdots & 0 \end{bmatrix}$$

(В случае мультиграфа элементы нулевых блоков – пустые множества.)

Определение 5.3. *Матрица инцидентности* простого графа G – это прямоугольная матрица $B = \left[b_{i,j}\right]$ размера $n \times m$, где $n = |V(G)|$, $m = |E(G)|$, в которой элемент $b_{i,j}$ равен 1, если вершина v_i инцидентна ребру e_j, и 0 в противном случае.

Из определения следует, что сумма элементов i-той строки матрицы инцидентности равна степени вершины v_i, а столбцы содержат по две единицы.

Строки матрицы B называют *векторами инцидентности* и обозначают как B_i.

§6. Представление графов в памяти компьютера

Пусть дан граф G, причём $|V(G)| = n$, $|E(G)| = m$.

Представление графа в виде матрицы смежности предполагает использование двух структур данных:
1. матрица A размера $n \times n$, при этом элемент $a_{i,j}$ матрицы равен некоторому специальному значению nil, если вершины v_i и v_j не являются смежными, в противном случае $a_{i,j}$ содержит атрибут ребра, соединяющего вершины v_i и v_j;
2. массив размера n атрибутов вершин.

Замечание 6.1. Для неориентированного графа возможно хранить не всю матрицу A, а только её половину, так как матрица – симметричная.

Замечание 6.2. Если G – мультиграф, то $a_{i,j}$ содержит список (или массив) атрибутов рёбер, соединяющих вершины v_i и v_j.

Свойства представления простого графа в виде матрицы смежности:

Размер в памяти	$O\left(n^2\right)$	
Время определения смежности двух вершин	$O\left(1\right)$	Достаточно посмотреть в $a_{i,j}$
Время нахождения всех рёбер, инцидентных вершине	$O\left(n\right)$	Нужно пройти по строке или столбцу матрицы
Время добавления новой вершины	$O\left(n^2\right)$	Придётся создавать новую матрицу и копировать в неё содержимое старой матрицы
Время удаления вершины		
Время добавления нового ребра	$O\left(1\right)$	Достаточно изменить $a_{i,j}$
Время удаления ребра		

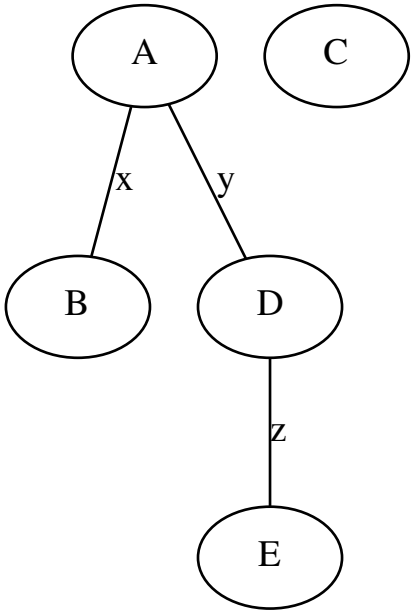
Свойства представления мультиграфа в виде матрицы смежности:

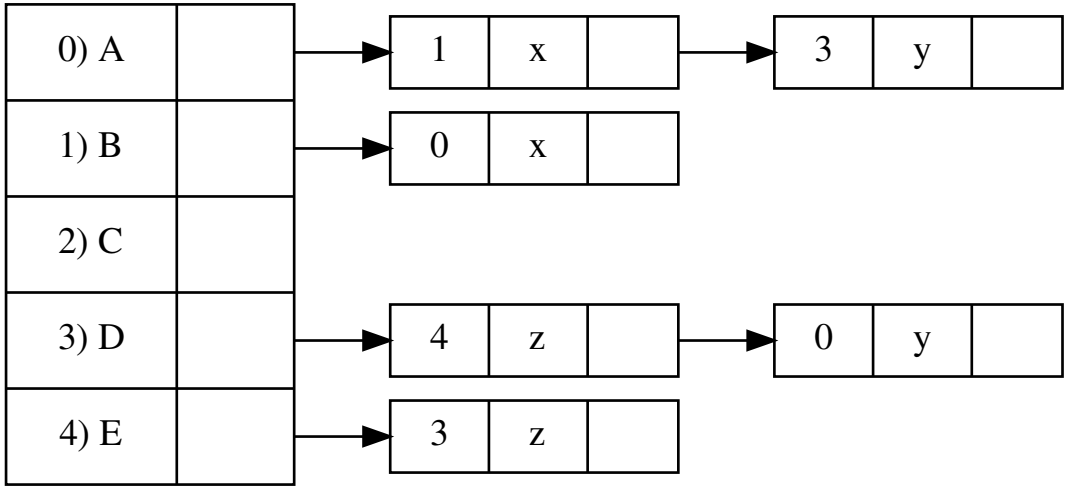
Размер в памяти	$O\left(n^2 + m\right)$
Время определения смежности двух вершин	$O\left(1\right)$
Время нахождения всех рёбер, инцидентных вершине	$O\left(\max\left\{n, m\right\}\right)$
Время добавления новой вершины	$O\left(n^2\right)$
Время удаления вершины	
Время добавления нового ребра	$O\left(m\right)$ – нужно перебрать все рёбра, соединяющие заданные две вершины
Время удаления ребра	

Замечание 6.3. Если предполагается частое удаление рёбер из мультиграфа, атрибуты рёбер лучше хранить в виде списка, а не в виде массива.

Представление (мульти)графа в виде списка инцидентности означает, что множество вершин графа хранится в виде массива структур размера n. i-тый элемент массива соответствует вершине v_i и содержит её атрибут, а также указатель на список инцидентных рёбер.
Для каждого ребра $\left\langle \left\{ v_i, v_j \right\}, a \right\rangle$, инцидентного вершине v_i, заводится отдельный элемент списка, содержащий номер вершины v_j, атрибут ребра и указатель на следующий элемент списка.

Пример.





Свойства представления (мульти)графа в виде списка инцидентности:

Размер в памяти	$O(n+m)$	
Время определения смежности двух вершин	$O(m)$	Нужно перебрать список рёбер
Время нахождения всех рёбер, инцидентных вершине	$O(1)$	Список инцидентных рёбер непосредственно доступен из структуры, описывающей вершину
Время добавления новой вершины	$O(n)$	Придётся создавать новый массив вершин
Время удаления вершины	$O(n+m)$	Придётся создавать новый массив вершин и перебирать списки рёбер
Время добавления нового ребра	$O(1)$	Нужно добавить один или два элемента списка
Время удаления ребра	$O(m)$	Нужно перебрать один или два списка рёбер

Замечание 6.4. В списке инцидентных рёбер, растущем из структуры, описывающей вершину простого графа, все рёбра оканчиваются в разных вершинах. Поэтому этот список можно считать списком смежных вершин, и представление простого графа в виде списка инцидентности можно назвать представлением в виде списка смежности.

Замечание 6.5. Если граф – связный, то необязательно хранить структуры, описывающие вершины, в массиве. Структуры могут быть расположены в памяти как угодно, при этом в списках инцидентных рёбер вместо номеров вершин нужно использовать указатели на вершины. Это позволяет свести время добавления новой вершины к $O(1)$, а время удаления вершины – к $O(m)$.

Замечание 6.6. Представление графа в виде матрицы смежности оправдано в случае плотного графа. В случае разреженного графа лучше использовать список инцидентности.

§7. Остовные деревья

Утверждение 7.1. *Любые две вершины связного графа, не содержащего циклов, соединяются единственной простой цепью.*

▷ Доказательство утверждения тривиально: если бы некоторые вершины u и v соединялись двумя различными простыми цепями, то согласно утверждению 3.2 граф содержал бы циклы. ◁

Напомним, что *дерево* – это связный граф с минимальным количеством рёбер: $|E(G)| = |V(G)| - 1$.

Утверждение 7.2. *Чтобы связный граф был деревом, необходимо и достаточно, чтобы он не содержал циклов.*

▷ Необходимость. Предположим, что в дереве T имеется цикл. Удалив из этого цикла одно ребро, мы не нарушим связность дерева, но при этом количество рёбер уменьшится. Значит, T – не дерево.

Достаточность. Доказывать будем индукцией по числу рёбер графа. Можно показать, что связные графы с количеством рёбер 0, 1 и 2, не содержащие циклов, являются деревьями.

Пусть любой не содержащий циклов связный граф с количеством рёбер, меньше или равным m, является деревом.

Рассмотрим не содержащий циклов связный граф G с $m+1$ рёбер.

Пусть произвольные вершины u и v графа соединены простой цепью, содержащей некоторое ребро e. Согласно утверждению 7.1, эта цепь – единственна. Поэтому, лишившись ребра e, граф станет несвязным, и разобьётся на два компонента с количеством рёбер m_1 и m_2, где $m_1 + m_2 = m$. Согласно гипотезе индукции, эти компоненты – деревья, поэтому количество вершин в них равно $m_1 + 1$ и $m_2 + 1$, соответственно.

Значит в графе G ровно $m_1 + m_2 + 2 = m + 2$ вершин, что на единицу больше количества в нём рёбер. Поэтому G – дерево. $\triangleleft$

Следствие из утверждения 7.2. Добавив к дереву T ребро e, мы получим связный граф G, в котором есть ровно один цикл, и этот цикл содержит ребро e.

$\triangleright$ Так как T – связный граф, то и G – связный граф. При этом G – не дерево, так как в нём на одно ребро больше, чем в дереве T. Поэтому, согласно утверждению 7.2, G содержит некоторый цикл.

Предположим, что e не входит в этот цикл. Тогда получается, что цикл состоит из рёбер дерева T, что невозможно, потому что, согласно утверждению 7.2, дерево не содержит циклов.

Предположим, что $e = \{u, v\}$ входит сразу в два разных цикла. Значит вершины u и v в дереве T соединяются сразу двумя цепями, что противоречит утверждению 7.1. $\triangleleft$

Утверждение 7.3. *Любой не содержащий циклов граф G, имеющий n вершин и $n-1$ рёбер, является деревом.*

$\triangleright$ Предположим, что G – несвязный. Тогда он разбивается на $k > 1$ компонент, имеющих m_1, m_2, ..., m_k рёбер. Причём $\sum\limits_{i=1}^{k} m_i = n - 1$.

Так как G не содержит циклов, то, согласно утверждению 7.2, каждый компонент является деревом и имеет $m_i + 1$ вершин. Поэтому получается, что общее количество вершин графа G равно $\sum\limits_{i=1}^{k} (m_i + 1) = n - 1 + k > n$.

Поэтому G – связный, и, согласно утверждению 7.2, он является деревом. $\triangleleft$

Напомним, что *остовным* называется подграф $G' = \langle V', E' \rangle$, включающий в себя все вершины надграфа $G = \langle V, E \rangle$ и часть рёбер: $V = V'$, $E' \subseteq E$.

Определение 7.1. *Остовное дерево* графа G – это остовный подграф графа G, являющийся деревом.

Следствие из утверждения 7.3. Любой не содержащий циклов остовный подграф графа G, имеющий n вершин и $n - 1$ рёбер, является остовным деревом.

▷ Доказательство непосредственно следует из утверждения 7.3. ◁

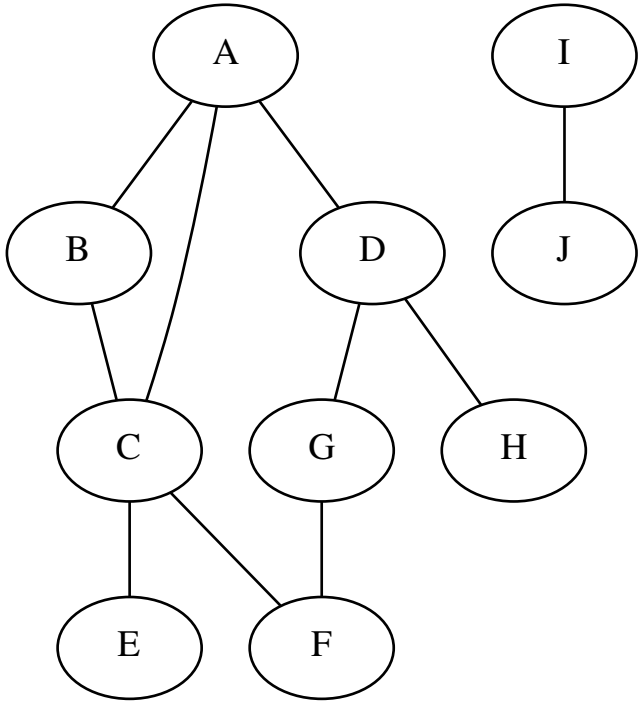
Тем самым, если нам удастся выбрать ровно $n-1$ не образующих циклов рёбер графа, имеющего n вершин, мы построим для этого графа остовное дерево. В противном случае граф несвязен, и для него невозможно построить остовное дерево.

§8. Обход графа

Обход – это основа, на которой строится большое количество алгоритмов, выполняющих анализ графов.

В некотором смысле, обход графа является аналогом перебора элементов массива, списка или дерева, приспособленным для работы с графом. При обходе вершины графа рассматриваются одна за другой в соответствии с принятым порядком обхода. При этом, как правило, ни одна вершина не обрабатывается дважды.

Пример.



Пусть вершины графа рассматриваются в следующем порядке:
A, B, C, E, F, G, D, H, I, J.
Это – один из возможных вариантов обхода графа *в глубину*.
Ещё один вариант обхода в глубину может выглядеть как
E, C, A, B, D, G, F, H, J, I
Если мы будем обрабатывать вершины в порядке
A, B, C, D, E, F, G, H, I, J,
то получится обход графа *в ширину*.

Рассмотрим схему работы алгоритма обхода графа в глубину (Depth-First Search – DFS) от вершины v.

1. Изначально все вершины графа *белые.*

2. Начиная с вершины v, мы строим в графе маршрут, последовательно добавляя к нему рёбра, инцидентные хотя бы одной белой вершине. При этом после добавления каждого ребра мы помечаем инцидентные этому ребру вершины *серым* цветом. Построение маршрута заканчивается в тот момент, когда мы попадаем в *тупик*, т.е. в вершину, у которой нет белых смежных вершин.

3. Возвращаемся по построенному маршруту до первой вершины u, у которой есть хотя бы одна белая смежная вершина. В процессе возвращения помечаем все вершины, у которых нет белых смежных вершин, *чёрным* цветом. Переходим к пункту 2 для $v = u$.

Мы будем считать, что структура, представляющая вершину в памяти, имеет поле $mark$, показывающее, каким цветом помечена данная вершина.

Если граф – несвязный, то запуск обхода от произвольной вершины v позволит посетить все вершины из компоненты связности, которой принадлежит вершина v.

Если мы хотим обойти все вершины несвязного графа, нам надо последовательно запускать обход от каждой непосещённой (т.е. белой) вершины.

Алгоритм обхода графа в глубину имеет сложность $O(|V| + |E|)$.

Алгоритм обхода графа в глубину:

```
1  DFS(in G)
2        for each v ∈ V(G):
3              v.mark ←white
4        for each v ∈ V(G):
5              if v.mark =white:
6                    VisitVertex(G, v)

8  VisitVertex(in G, in v)
9        v.mark ←gray
10       Действия при заходе в вершину v
11       for each ⟨{v, u}, a⟩ ∈ E(G):
12             if u.mark =white:
13                   Действия при проходе по ребру ⟨{v, u}, a⟩
14                   VisitVertex(G, u)
15       v.mark ←black
16       Действия перед выходом из вершины v
```

Нерекурсивный алгоритм обхода графа в глубину:

```
1  DFS(in G)
2     for each v ∈ V(G):
3        v.mark ←white
4     InitStack(out s)
5     for each w ∈ V(G):
6        if w.mark =white:
7           Push(s, w)
8           while not StackEmpty(s):
9              v ←Pop(s)
10             if v.mark =white:
11                v.mark ←gray
12                Действия при заходе в вершину v
13                Push(s, v)
14                for each ⟨{v, u}, a⟩ ∈ E(G):
15                   if u.mark =white:
16                      Push(s, u)
17             else if v.mark =gray:
18                v.mark ←black
19                Действия перед выходом из вершины v
```

Алгоритм обхода графа в ширину (Breadth-First Search – BFS) от вершины v сначала посещает все вершины, смежные с вершиной v, затем все вершины, смежные со смежными вершинами, и т.д.

Аналогично обходу в глубину, все посещаемые вершины помечаются, но не цветом, а булевским значением true. Соответственно, изначально все вершины помечены значением false.

В случае несвязного графа обход запускается от каждой непосещённой вершины.

Рекурсивный алгоритм обхода в ширину нерационален, поэтому применяется итерационный алгоритм, аналогичный нерекурсивному алгоритму обхода в глубину, но использующий не стек, а очередь.

Алгоритм обхода графа в ширину работает за время $O(|V| + |E|)$.

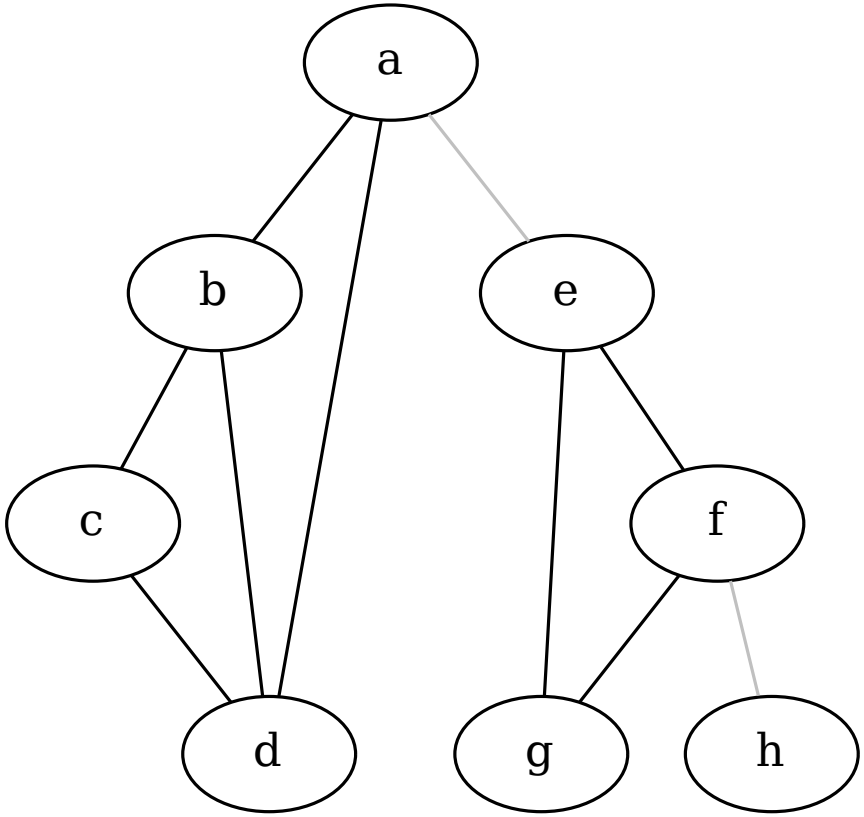
Алгоритм обхода графа в ширину:

```
1  BFS(in G)
2      for each v ∈ V(G):
3          v.mark ← false
4      InitQueue(out q)
5      for each w ∈ V(G):
6          if not w.mark:
7              w.mark ← true
8              Enqueue(q, w)
9              while not QueueEmpty(q):
10                 v ← Dequeue(q)
11                 Действия при заходе в вершину v
12                 for each ⟨{v, u}, a⟩ ∈ E(G):
13                     if not u.mark:
14                         u.mark ← true
15                         Enqueue(q, u)
```

§9. Компоненты рёберной двусвязности

Определение 9.1. *Мост* в простом графе – это ребро, после удаления которого увеличивается число компонент связности графа.

Пример. (мосты помечены серым)



Утверждение 9.1. В связном простом графе нет мостов тогда и только тогда, когда между любыми двумя вершинами этого графа существуют хотя бы две не имеющие общих рёбер цепи.

▷ Необходимость. Предположим, что в не имеющем мостов связном графе существуют две вершины u и v такие, что все соединяющие их цепи имеют хотя бы одно общее ребро. Тогда удаление этого ребра приведёт к разбиению графа на две компоненты.

Достаточность. Пусть любые две вершины связного графа соединяются хотя бы двумя не имеющими общих рёбер цепями. Тогда удаление любого ребра графа приведёт к разрыву не более одной из этих цепей. ◁

Определение 9.2. Компоненты связности, получающиеся в простом графе после удаления всех мостов, называются *компонентами рёберной двусвязности*.

Алгоритм построения компонент рёберной двусвязности простого графа включает два обхода графа в глубину. Приведём схему работы алгоритма.

1. В каждой компоненте связности графа произвольно выбирается *главная вершина*. Из неё стартует первый обход графа в глубину, в результате которого формируется ориентированное остовное дерево компоненты связности с корнем в главной вершине.
   Кроме того, все вершины компоненты помещаются в очередь $queue$ в том порядке, в котором они были посещены.

2. Последовательно выбираем из очереди $queue$ вершины графа и, если очередная вершина v ещё не участвовала во втором обходе, запускаем из неё второй обход графа в глубину. Этот обход имеет одну особенность: через ребро, принадлежащее остовному дереву, можно проходить только в направлении к его корню.
   Все вершины, пройденные в процессе второго обхода графа из вершины v, будут принадлежать одной компоненте рёберной двусвязности.

```
1  DFS1(in G, in/out queue)
2       for each v ∈ V(G):
3            v.mark ←white
4       for each v ∈ V(G):
5            if v.mark =white:
6                 v.parent ←NULL
7                 VisitVertex1(G, v, in/out queue)

9  VisitVertex1(in G, in v, in/out queue)
10      v.mark ←gray
11      Enqueue(queue, v)
12      for each ⟨{v, u}, a⟩ ∈ E(G):
13           if u.mark =white:
14                u.parent ← v
15                VisitVertex1(G, u, in/out queue)
16      v.mark ←black
```

```
1  DFS2(in G, in queue)
2        for each v ∈ V(G):
3              v.comp ← −1
4        component ← 0
5        while not QueueEmpty(queue):
6              v ←Dequeue(queue)
7              if v.comp = −1:
8                    VisitVertex2(G, v, component)
9                    component ← component + 1

11 VisitVertex2(in G, in v, in component)
12       v.comp ← component
13       for each ⟨{v, u}, a⟩ ∈ E(G):
14             if u.comp = −1 and u.parent ≠ v:
15                   VisitVertex2(G, u, component)
```

§10. Алгоритм Крускала

В принципе, остовное дерево легко получить в процессе обхода графа в глубину или в ширину, но мы рассмотрим другой способ, который пригодится для алгоритма Крускала.

Пусть имеется последовательность $\{e[i]\}$ рёбер графа G, имеющего n вершин.

Будем формировать множество рёбер E' остовного дерева следующим образом: до тех пор, пока размер E' меньше $n-1$, будем просматривать последовательность $\{e[i]\}$ и добавлять i-тое ребро в E', если оно не образует циклов с рёбрами, уже добавленными в E'.

Для конкретного графа можно построить не одно остовное дерево. Какие именно рёбра попадут в остовное дерево, зависит от того, в каком порядке они записаны в последовательности $\{e[i]\}$.

В общем случае в процессе формирования множества E' мы будем получать несвязные остовные подграфы графа G: $G_0, G_1, G_2, \ldots G_{n-2}$, и только финальный подграф G_{n-1} точно будет связным, так как по следствию из утверждения 7.3 он будет являться остовным деревом.

Обратите внимание на то, что G_0 соответствует пустому множеству E', т.е. вообще не содержит рёбер.

Чтобы в процессе работы алгоритма эффективно отбрасывать рёбра, образующие циклы с уже добавленными рёбрами, мы будем использовать лес непересекающихся множеств, в узлами которого будут являться вершины графа.

Алгоритм формирования множества рёбер E' остовного дерева по последовательности $\{e[i]\}$ рёбер графа:

```
1  SpanningTree(in G, in {e[i]}_0^{m-1}, out E')
2        E' ← ∅
3        i ← 0
4        while i < m and |E'| < |V(G)| − 1:
5              let e[i] = {u, v}:
6                    if Find(u) ≠ Find(v):
7                          E' ← E' ∪ e[i]
8                          Union(u, v)
9              i ← i + 1
10       if |E'| ≠ |V(G)| − 1: panic "Несвязный граф"
```

Предполагается, что перед началом работы алгоритма каждая вершина графа является отдельным деревом в лесу непересекающихся множеств.

Время работы алгоритма – $O(m\alpha(n))$.

Рассмотрим связный мультиграф, размеченный вещественными или целыми числами. То есть каждому ребру в таком мультиграфе соответствует свой *вес*. Вес ребра e будем обозначать как $w(e)$.

Определение 10.1. *Минимальное остовное дерево* связного мультиграфа – это остовное дерево, сумма весов рёбер которого минимальна.

Алгоритм Крускала построения минимального остовного дерева предполагает, что рёбра мультиграфа сортируются по возрастанию их весов, после чего вызывается рассмотренный алгоритм SpanningTree:

```
1 MST_Kruskal(in G, out T)
2         V(T) ← V(G)
3         e ←Sort(E(G))
4         SpanningTree(e, |V(G)|, out E(T))
```

Время работы алгоритма Крускала определяется временем сортировки рёбер, т.е. в общем случае оно равно $O(m \lg m)$, где m – число рёбер.

Утверждение 10.1. Остовные подграфы $G_0, G_1, \ldots, G_{n-1}$ связанного мультиграфа G, получающиеся в процессе работы алгоритма Крускала, являются подграфами минимального остовного дерева мультиграфа G.

▷ Будем доказывать это утверждение индукцией по номеру подграфа G_i (другими словами, по количеству рёбер в подграфе).

Для пустого подграфа G_0 утверждение выполняется тривиально.

Пусть G_i является подграфом некоторого минимального остовного дерева. Докажем, что при добавлении к G_i очередного ребра e с минимальным среди оставшихся рёбер весом, не образующего циклов с рёбрами G_i, получается G_{i+1}, который тоже является подграфом некоторого минимального остовного дерева.

Предположим, что e не входит ни в одно минимальное остовное дерево T, подграфом которого является G_i. Добавим e в T. По следствию из утверждения 7.2 мы получим единственный цикл c, и этот цикл будет содержать ребро e. Заметим, что цикл c не входит в G_i, потому что при добавлении e к G_i мы циклов не получаем. Поэтому c обязательно содержит рёбра, не входящие в G_i. Вес каждого из этих рёбер по крайней мере не меньше, чем $w(e)$, потому что $w(e)$ – минимальный среди весов рёбер, не входящих в G_i.

Удалив из цикла c одно ребро с весом, не меньшим, чем $w(e)$, мы получим граф T'. Этот граф имеет такое же число вершин и рёбер, что и дерево T, и при этом не содержит циклов, так как цикл c – единственный. Согласно утверждению 7.3, T' является деревом. При этом сумма весов рёбер дерева T' не превышает суммы весов рёбер дерева T. Значит либо T не является минимальным, либо T', содержащее e, тоже является минимальным. В любом случае приходим к противоречию. $\triangleleft$

§11. Алгоритм Прима

В отличие от алгоритма Крускала, алгоритм Прима строит минимальное остовное дерево мультиграфа G путём последовательного наращивания единственного древовидного фрагмента, то есть в процессе работы алгоритма мы будем получать последовательность деревьев $T_0, T_1, \ldots, T_{n-1}$, в которой T_0 – пустое дерево, содержащее одну произвольно выбранную вершину G, а T_{n-1} – минимальное остовное дерево мультиграфа G.

На каждом шаге алгоритм Прима добавляет к T_i ребро с минимальным весом из числа рёбер, соединяющих одну из вершин из T_i с вершиной графа, не принадлежащей T_i.

Алгоритм Прима использует для поиска ребра с минимальным весом очередь с приоритетами, в которой содержатся вершины мультиграфа, не принадлежащие T_i, но к которым ведут рёбра от вершин, принадлежащих T_i.

Каждая вершина графа при этом представляется тройкой $\langle index, key, value \rangle$, в которой:

- $index$ содержит номер вершины в пирамиде, через которую реализована очередь с приоритетами;
- key равен весу ребра, соединяющего вершину с одной из вершин, принадлежащих T_i (если таких рёбер несколько, выбирается ребро с минимальным весом);
- $value$ хранит другой конец вышеупомянутого ребра.

Поле $index$ у вершин, не входящих в очередь с приоритетами, равно -2, если вершина уже принадлежит T_i, и -1 – в противном случае.

Алгоритм Прима:

```
1  MST_Prim(in G, out T)
2      for each v ∈ V(G):
3          v.index ← −1
4      V(T) ← V(G), E(T) ← ∅
5      InitPriorityQueue(out q, |V(G)| − 1)
6      let v ∈ V(G):
7          loop:
8              v.index ← −2
9              for each ⟨{v, u}, a⟩ ∈ E(G):
10                 if u.index = −1:
11                     u.key ← a, u.value ← v
12                     Insert(q, u)
13                 else if u.index ≠ −2 and a < u.key:
14                     u.value ← v
15                     DecreaseKey(q, u.index, a)
16             if QueueEmpty(q): break
17             v ← ExtractMin(q)
18             E(T) ← E(T) ∪ ⟨{v, v.value}, v.key⟩
```

Время работы: $O(m \lg n + n \lg n)$, где $n = |V(G)|$, $m = |E(G)|$.

Утверждение 11.1. Деревья $T_0, T_1, \ldots, T_{n-1}$, получающиеся в процессе работы алгоритма Прима для мультиграфа G, являются поддеревьями минимального остовного дерева мультиграфа G.

Докажите самостоятельно!

§12. Ориентированные графы

Определение 12.1. Пара $\langle V, E\rangle$ называется *простым орграфом*, если V – непустое конечное множество элементов, называемых *вершинами*, а E – множество упорядоченных пар различных элементов из V, называемых *дугами*.

Орграф можно рассматривать как пару $\langle V, \Gamma\rangle$, где $\Gamma \subseteq V^2$ – отображение, заданное на множестве V. Тогда $\Gamma(v)$ – множество конечных вершин всех дуг, *исходящих* из вершины v, а $\Gamma^{-1}(v)$ – множество начальных вершин всех дуг, входящих в v.

При описании орграфов используют те же понятия, термины и характеристики, что и для неориентированных графов. Однако орграфы имеют свою специфику, обусловленную наличием ориентации дуг. Например, для каждой вершины v орграфа наряду с её степенью $\deg v$ определены *полустепень исхода* $\deg^+ v = |\Gamma(v)|$ и *полустепень захода* $\deg^- v = \left|\Gamma^{-1}(v)\right|$. При этом $\deg v = \deg^+ v + \deg^- v$ и $\sum\limits_{u \in V(G)} \deg^+ u = \sum\limits_{u \in V(G)} \deg^- u = |E(G)|$.

Определение 12.2. *Матрица смежности* простого орграфа G – это квадратная матрица $A = \left[a_{i,j}\right]$ порядка $n = |V(G)|$, в которой элемент $a_{i,j}$ равен 1, если в орграфе есть дуга $\left\langle v_i, v_j \right\rangle$, и 0, если такой дуги нет.

Из определения следует, что сумма элементов i-той строки матрицы смежности равна полустепени исхода вершины v_i, а сумма элементов j-того столбца равна полустепени захода вершины v_j.

Определение 12.3. *Матрица инцидентности* простого орграфа G – это прямоугольная матрица $B = \left[b_{i,j}\right]$ размера $n \times m$, где $n = |V(G)|$, $m = |E(G)|$, в которой элемент $b_{i,j}$:
– равен 1, если вершина v_i является начальной вершиной дуги e_j;
– равен -1, если вершина v_i является конечной вершиной дуги e_j;
– равен 0, если вершина v_i и дуга e_j не инцидентны.

Ориентированный мультиграф определяется аналогично неориентированному.

Определение 12.4. *Ориентированным маршрутом* в орграфе называется конечная последовательность дуг, в которой для любых двух последовательных дуг a и b справедливо, что a входит в вершину, из которой исходит b. При этом *длина ориентированного маршрута* – это количество принадлежащих ему дуг.

Нетрудно заметить, что все дуги в ориентированном маршруте направлены от начальной вершины к конечной. Если разрешить «свободную» ориентацию дуг, то получится *полумаршрут* – аналог маршрута в неориентированном графе.

Определение 12.5. Ориентированный маршрут называется *ориентированной цепью*, если все принадлежащие ему дуги различны, и *ориентированной простой цепью*, если в добавок любая вершина графа инцидентна не более, чем двум дугам маршрута.

Ориентированная простая цепь также называется *путём*. Кроме того, для определения слабо связного орграфа нам понадобится понятие *полупуть* – полумаршрут, в котором различны все дуги и вершины (кроме, возможно, начальной и конечной).

Говорят, что вершина u *достижима* из вершины v, если существует путь $v \to \ldots \to u$. Если при этом v достижима из u, то u и v – *взаимно достижимы*. Мы будем использовать обозначение $u \sim v$, чтобы показать, что вершины u и v взаимно достижимы.

Определение 12.6. *Ориентированным циклом* (*ориентированной простым циклом*) называется ориентированная цепь (ориентированная простая цепь), начальная и конечная вершины которой совпадают.

Определение 12.7. Орграф называется *сильно связным*, если любые две вершины в нём взаимно достижимы.

Определение 12.8. Орграф называется *односторонне связным*, если в любой паре его вершин хотя бы одна достижима из другой.

Определение 12.9. Орграф называется *слабо связным*, если в нём любые две вершины соединены полупутём.

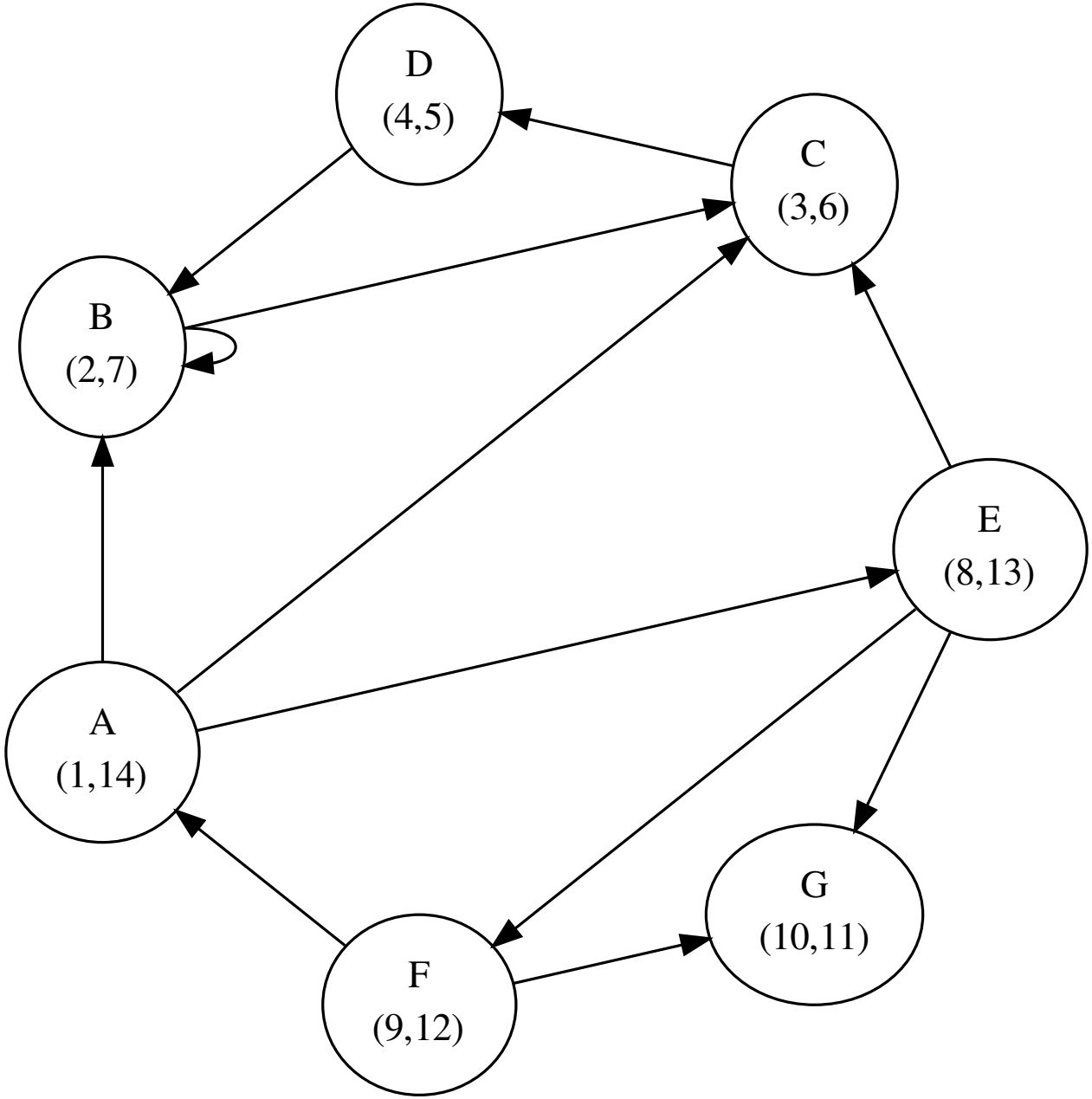
Кроме того, выделяют *ациклические орграфы*, не имеющие ни одного цикла.

Алгоритмы обхода вершин и рёбер графа легко распространяются и на орграфы, поэтому мы не будем их отдельно рассматривать.

§13. Классификация дуг орграфа

Рассмотрим обход вершин орграфа в глубину. При этом для каждой вершины будем записывать *время захода* $T1$ и *время выхода* $T2$:

```
1  global time ← 1

3  CalculateTimes(in G)
4      for each v ∈ V(G):
5          v.T1 ← v.T2 ← 0
6      for each v ∈ V(G):
7          if v.T1 = 0:
8              VisitVertex(G, v)

10 VisitVertex(in G, in v)
11     v.T1 ← time;   time ← time + 1
12     for each ⟨v, u⟩ ∈ E(G):
13         if u.T1 = 0:
14             VisitVertex(G, u)
15     v.T2 ← time;   time ← time + 1
```


D
(4,5)
C
(3,6)
B
(2,7)
E
(8,13)
A
(1,14)
G
(10,11)
F
(9,12)

Рассмотрим некоторую дугу $e = \langle u, v \rangle$ орграфа, вершины которого размечены временами захода и выхода. Пусть e – не петля.
Возможны три варианта соотношений между временани захода и выхода для вершин u и v:

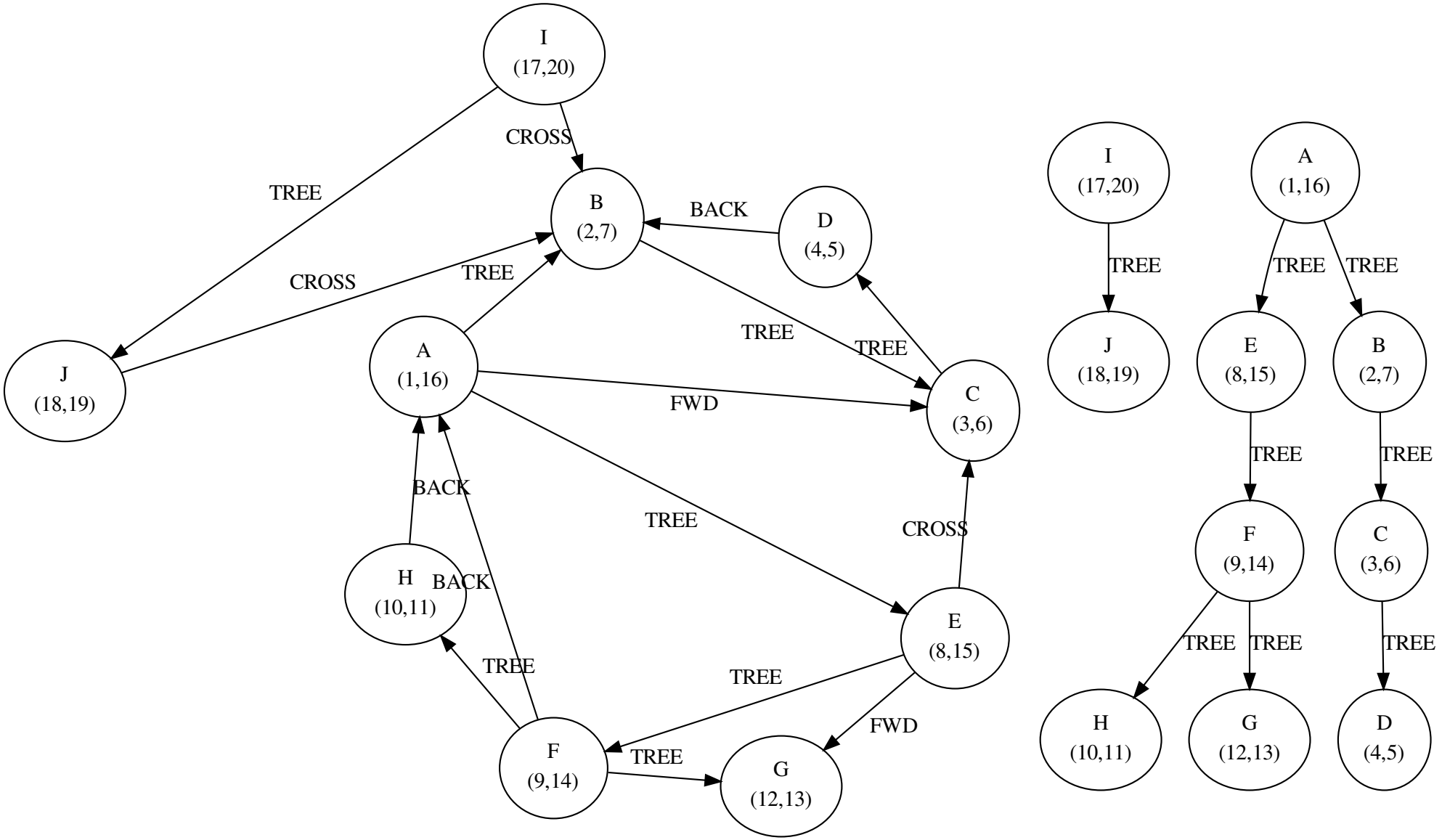
1. $u.T1 < v.T1 < v.T2 < u.T2$ – вершина v была посещена после захода в вершину u и до выхода из вершины u.
   Тогда если при обходе орграфа в глубину мы встретили дугу e в момент, когда вершина v была не помеченной, то дуга e – *древесная* (TREE). В противном случае дуга e – *прямая* (FWD).

2. $v.T1 < u.T1 < u.T2 < v.T2$ – вершина u была посещена после захода в вершину v и до выхода из вершины v. Тогда дуга e – *обратная* (BACK).

3. $v.T2 < u.T1$ – вершина u была посещена после выхода из вершины v. Тогда дуга e – *поперечная* (CROSS).

Все петли будем классифицировать как обратные дуги.

Пример. (Справа из графа удалены недревесные дуги)



Если оставить в орграфе только древесные дуги, получится так называемый *лес обхода в глубину*. Этот лес состоит из ориентированных деревьев.

Подграф леса обхода в глубину называется *поддеревом*, если он хотя бы слабо связен. Каждое поддерево имеет корень, из которого достижимы все остальные вершины поддерева.

По логике алгоритма обхода в глубину ясно, что время захода для корня любого поддерева меньше времени захода для всех остальных вершин поддерева, а время выхода – больше времени выхода для всех остальных вершин поддерева.

§14. Компоненты сильной связности

Отношение взаимной достижимости является отношением эквивалентности:

Рефлексивность. $u \sim u$, так как всегда существует путь $u \mapsto u$.

Симметричность. Если $u \sim v$, то существуют пути $u \mapsto v$ и $v \mapsto u$. Из этого следует $v \sim u$.

Транзитивность. Пусть $u \sim v$ и $v \sim w$. Тогда существуют пути $u \mapsto v$ и $v \mapsto w$, откуда следует существование пути $u \mapsto w$. Аналогично, существуют пути $w \mapsto v$ и $v \mapsto u$, откуда следует существование пути $w \mapsto u$. Поэтому $u \sim w$.

Любое отношение эквивалентности разбивает множество, на котором оно определено, на классы эквивалентности. Класс эквивалентности обозначается как $[x]$, где x – один из его элементов, и определяется следующим образом:

$$[x] = \{y \mid x \sim y\}$$

Легко доказать, что классы эквивалентности не пересекаются.

Определение 14.1. Подграфы, порождённые классами эквивалентности, на которые отношение взаимной достижимости разбивает множество вершин орграфа, называются *компонентами сильной связности*.

Несложно доказать, что если вершины u и v принадлежат одной компоненте связности, то все вершины, принадлежащие любому пути $u \mapsto v$ (а также $v \mapsto u$), также принадлежат этой компоненте.

Компонента сильной связности для вершины u вычисляется в два этапа.

На первом этапе мы выполняем обход графа (не важно, в ширину или в глубину) от вершины u и формируем множество вершин A, достижимых из вершины u.

На втором этапе мы меняем направление всех дуг графа на противоположное и опять запускаем обход графа от вершины u, формируя множество вершин B, из которых достижима вершина u.

Компонента сильной связности $[u] = A \cap B$.

Этот простой алгоритм работает за время $O(m)$, где m – количество дуг в графе.

В §13 мы выяснили, что в результате обхода орграфа в глубину получается лес, состоящий из ориентированных деревьев.

Можно показать, что компонента сильной связности, если оставить в ней только древесные дуги, образует поддерево леса обхода в глубину. Мы будем называть корень такого поддерева *корнем компоненты*.

Пусть в процессе обхода графа в глубину для каждого узла u вычисляется время захода в него $u.T1$. Так как компонента связности образует поддерево леса обхода в глубину, то время захода для корня компоненты будет меньше времени захода для любого другого узла той же компоненты.

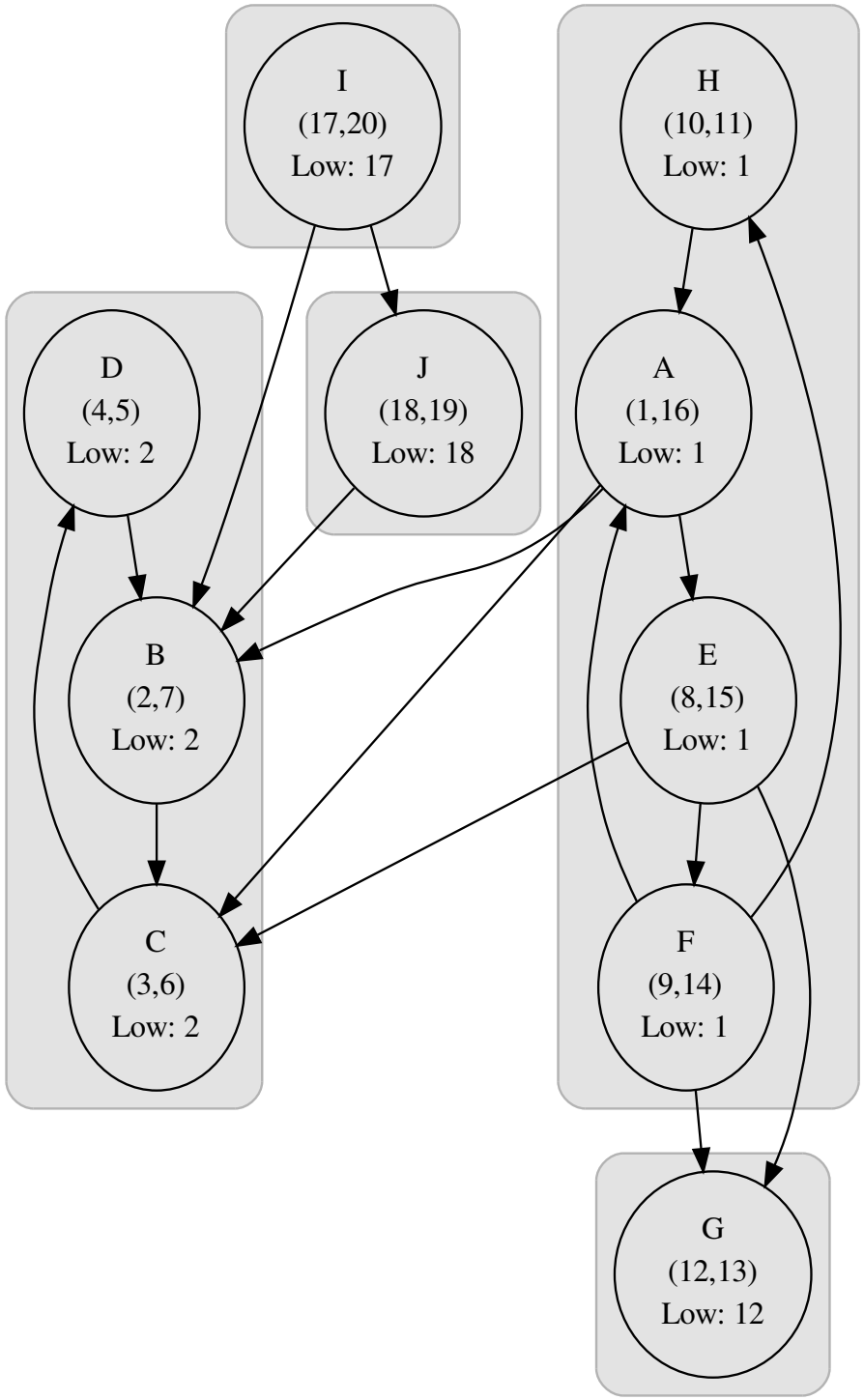
Алгоритм Тарьяна выполняет обход орграфа в глубину и вычисляет все компоненты сильной связности в том порядке, в котором завершается обход соответствующих им поддеревьев леса обхода в глубину. При этом компоненты нумеруются, и в поле $comp$ каждой вершины орграфа записывается номер её компоненты.

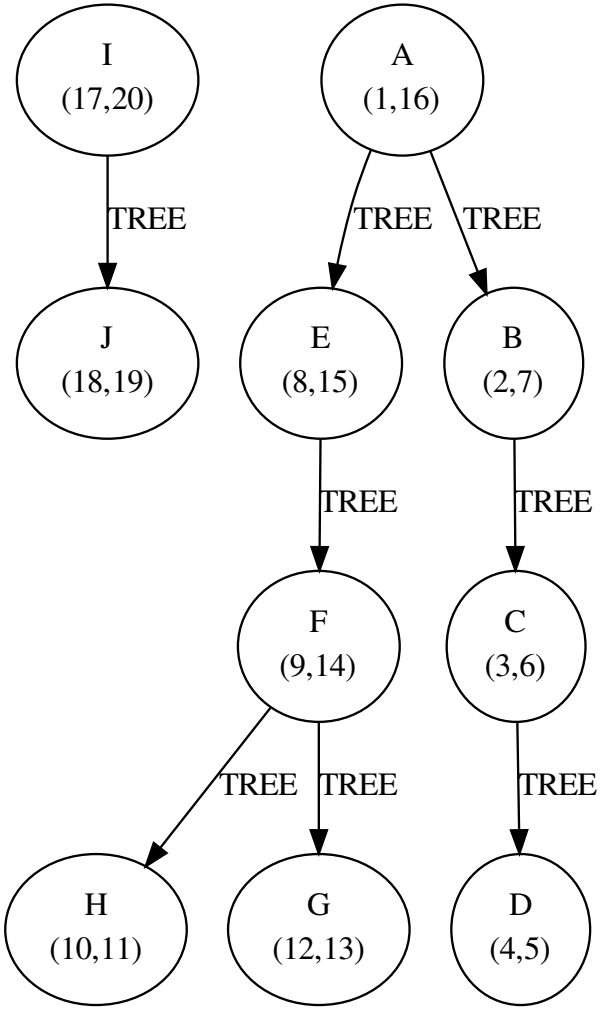
```
1  global time ← 1
2  global count ← 1

4  Tarjan(in G)
5      for each v ∈ V(G):
6          v.T1 ← v.comp ← 0
7      InitStack(out s, |V(G)|)
8      for each v ∈ V(G):
9          if v.T1 = 0:
10             VisitVertex_Tarjan(G, v, s)
```

Основная идея алгоритма Тарьяна заключается в том, что в процессе обхода для каждого узла u вычисляется значение $u.low$, равное минимальному времени захода для всех узлов, достижимых из u, но не принадлежащих ещё ни одной вычисленной компоненте.

При завершении обхода каждого поддерева с корнем в некотором узле u алгоритм смотрит, не равны ли значения $u.low$ и $u.T1$. Если они оказываются равны, значит u – корень новой компоненты.

```
12 VisitVertex_Tarjan(in G, in v, in/out s)
13     v.T1 ← v.low ← time;    time ← time + 1
14     Push(s, v)
15     for each ⟨v, u⟩ ∈ E(G):
16         if u.T1 = 0:
17             VisitVertex_Tarjan(G, u, s)
18         if u.comp = 0 and v.low > u.low:
19             v.low ← u.low
20     if v.T1 = v.low:
21         do: u ←Pop(s);    u.comp ← count  while u ≠ v
22         count ← count + 1
```

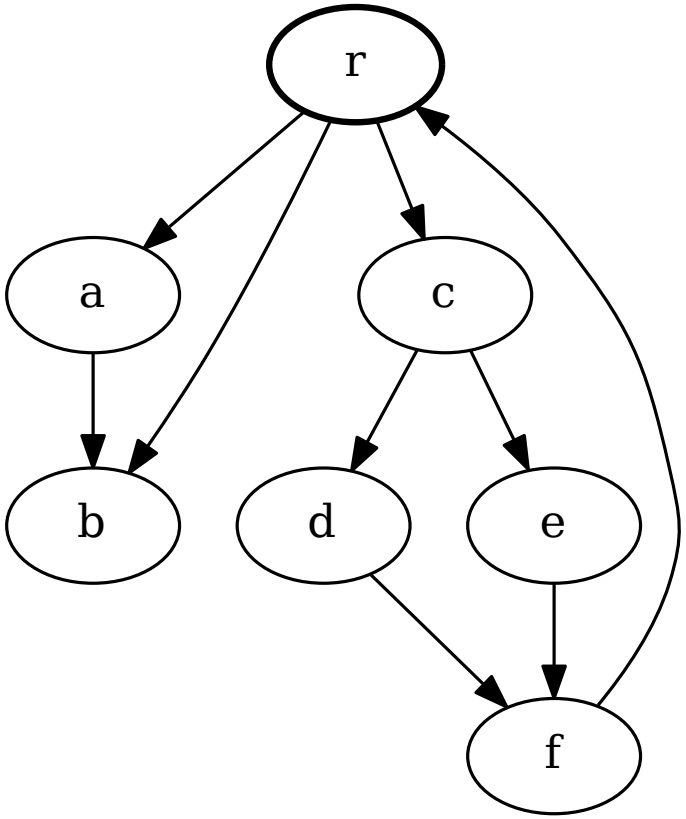

I
(17,20)
Low: 17
H
(10,11)
Low: 1
D
(4,5)
Low: 2
J
(18,19)
Low: 18
A
(1,16)
Low: 1
B
(2,7)
Low: 2
E
(8,15)
Low: 1
C
(3,6)
Low: 2
F
(9,14)
Low: 1
G
(12,13)
Low: 12


I
(17,20)
A
(1,16)
TREE
TREE
TREE
J
(18,19)
E
(8,15)
B
(2,7)
TREE
TREE
F
(9,14)
C
(3,6)
TREE
TREE
TREE
H
(10,11)
G
(12,13)
D
(4,5)

§15. Доминаторы в управляющих графах

Определение 15.1. *Управляющий граф* – это орграф $G = \langle V, E, r \rangle$ с выделенной начальной вершиной r, из которой достижима любая другая вершина этого орграфа. Начальную вершину в управляющем графе принято называть *входом* или *корнем*.

Пример.

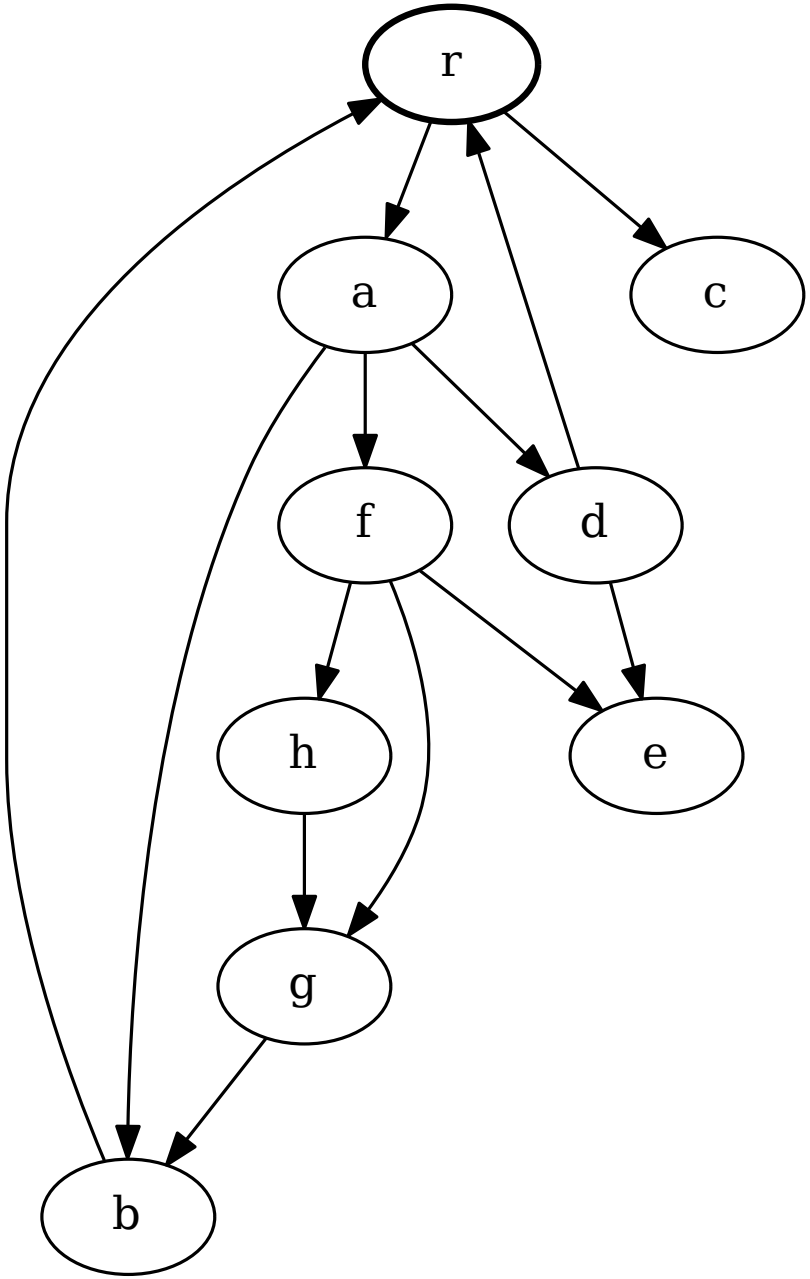


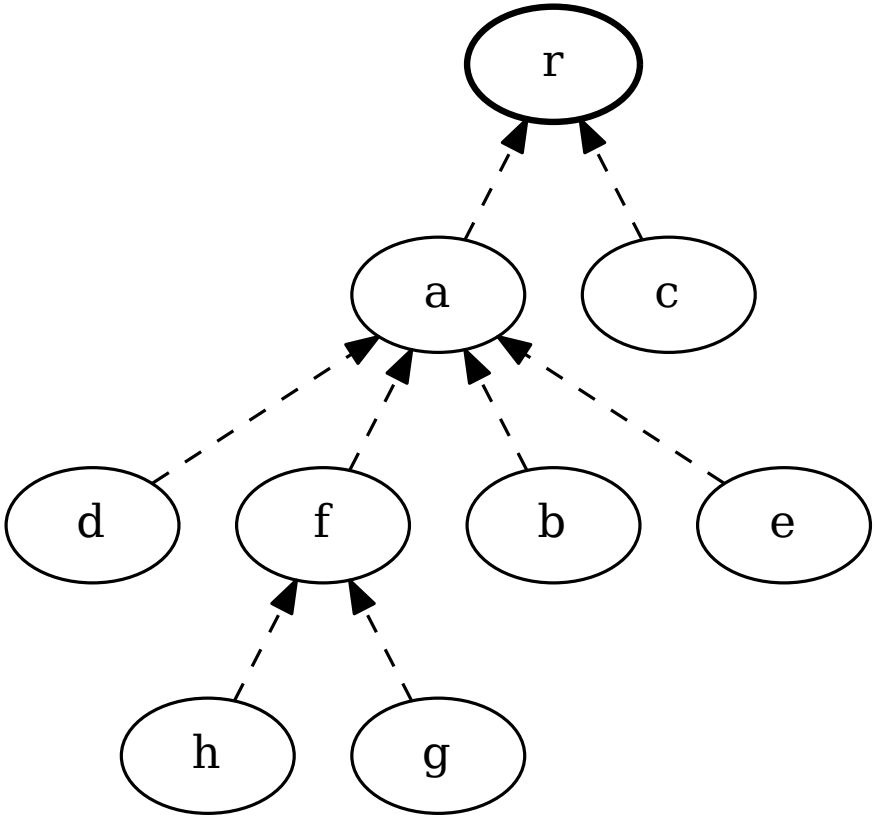
Определение 15.2. Говорят, что в управляющем графе $G = \langle V, E, r \rangle$ вершина v *доминирует* над вершиной u, если все пути из r в u проходят через v. Вершина v при этом называется *доминатором* вершины u.

Определение 15.3. Вершина v является *непосредственным доминатором* вершины u, если v доминирует над u, и любой другой доминатор вершины u доминирует над v.
Обозначение: $v = \mathrm{idom}(u)$.

Определение 15.4. *Дерево доминаторов* для управляющего графа $G = \langle V, E, r \rangle$ – это ориентированное дерево на множестве вершин V с корнем в r, в котором дуги соединяют вершины с их непосредственными доминаторами.

Пример. (Управляющий граф и его дерево доминаторов)





Простейший способ построения дерева доминаторов заключается в том, что сначала строится предпорядок вершин при обходе управляющего графа $G = \langle V, E, r \rangle$ в глубину от корня r, а затем для каждой вершины v из предпорядка выполняются следующие действия:

1. все вершины графа, кроме v, помечаются «белым» цветом, а вершина v помечается «чёрным» цветом;

2. выполняется обход графа в глубину от корня r (а можно и в ширину);

3. находятся вершины, оставшиеся «белыми»; вершина v становится непосредственным доминатором для этих вершин.

Заметим, что в процессе работы описанного алгоритма непосредственный доминатор вершины графа может несколько раз поменяться. Сложность алгоритма – $O(nm)$, где n и m – количество вершин и рёбер, соответственно.

Эффективный алгоритм построения дерева доминаторов был представлен Ленгауэром и Тарьяном в 1979 году.

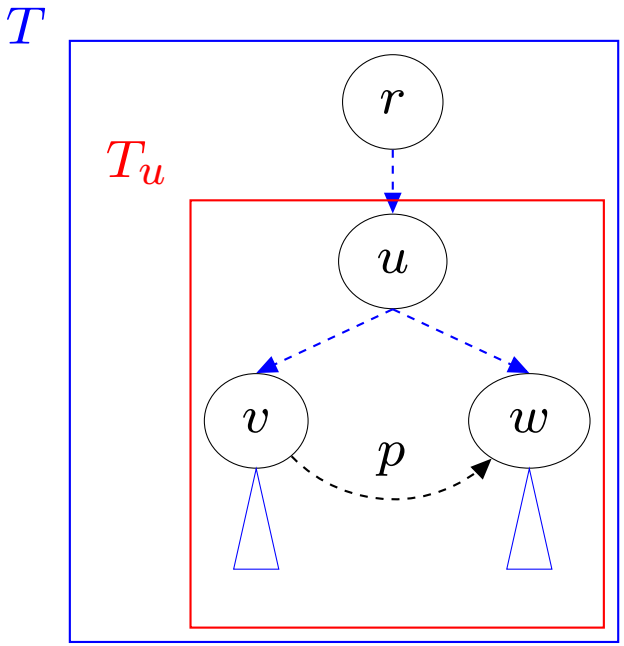
Прежде чем перейти к рассмотрению этого алгоритма, договоримся, что на управляющем графе $G = \langle V, E, r \rangle$, для которого мы будем строить дерево доминаторов, был выполнен обход в глубину от его корня r, в результате которого мы получили:

1. дерево обхода в глубину T с корнем в r, которое задаётся полями $parent$ в каждой вершине графа;

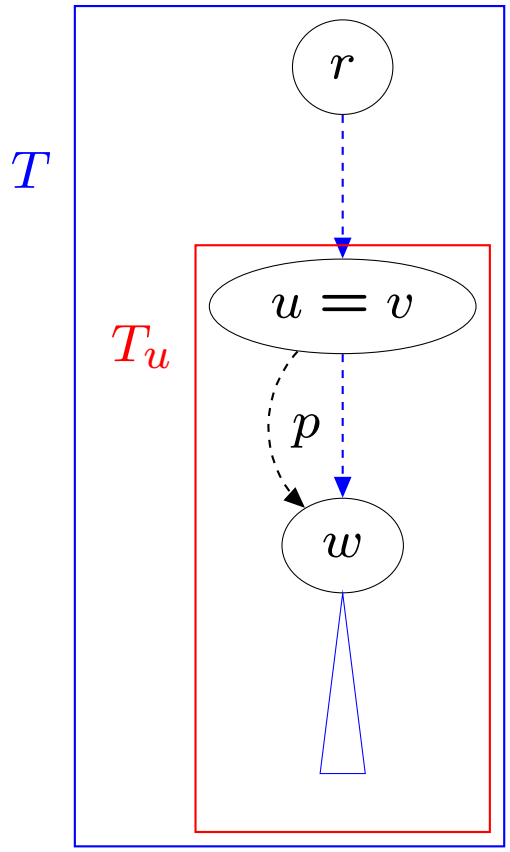
2. отношение строгого порядка $\prec$ на множестве вершин такое, что $v \prec w$ тогда и только тогда, когда при выполненном обходе в глубину время захода для вершины v меньше времени захода для вершины w.

Лемма 15.1. Если v и w – такие вершины орграфа G, что $v \prec w$, то любой путь p из v в w должен содержать общего предка вершин v и w в дереве T.

▷ Пусть T_u – самое маленькое поддерево дерева T, содержащее все вершины пути p, причём некоторая вершина u – корень этого поддерева. Отметим, что так как вершины v и w входят в T_u, то u – один из общих предков вершин v и w в дереве T.



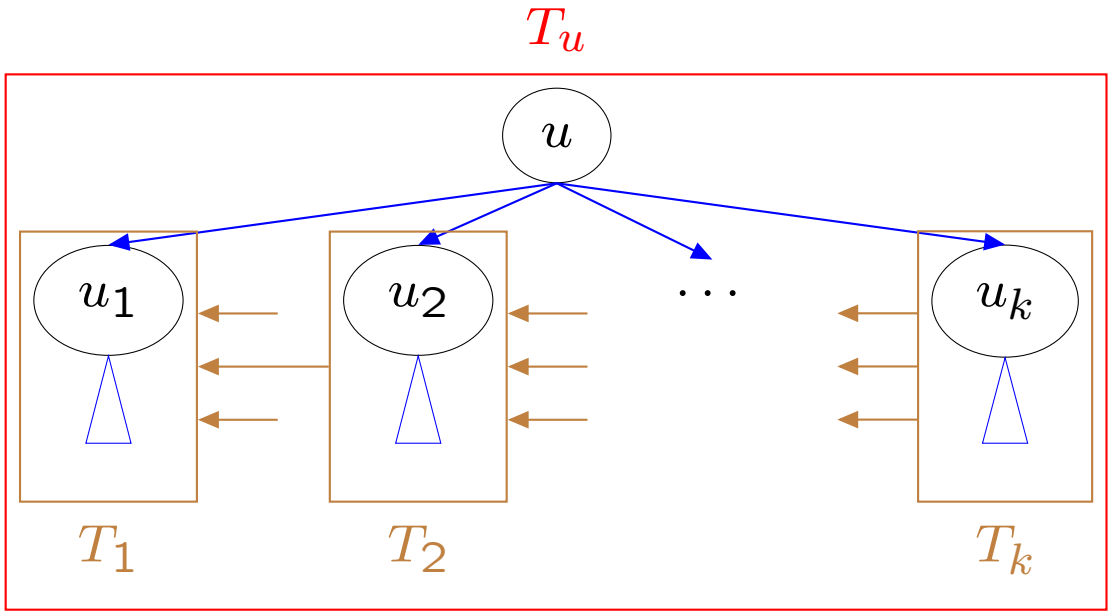
Если $u = v$, то лемма выполняется сразу.



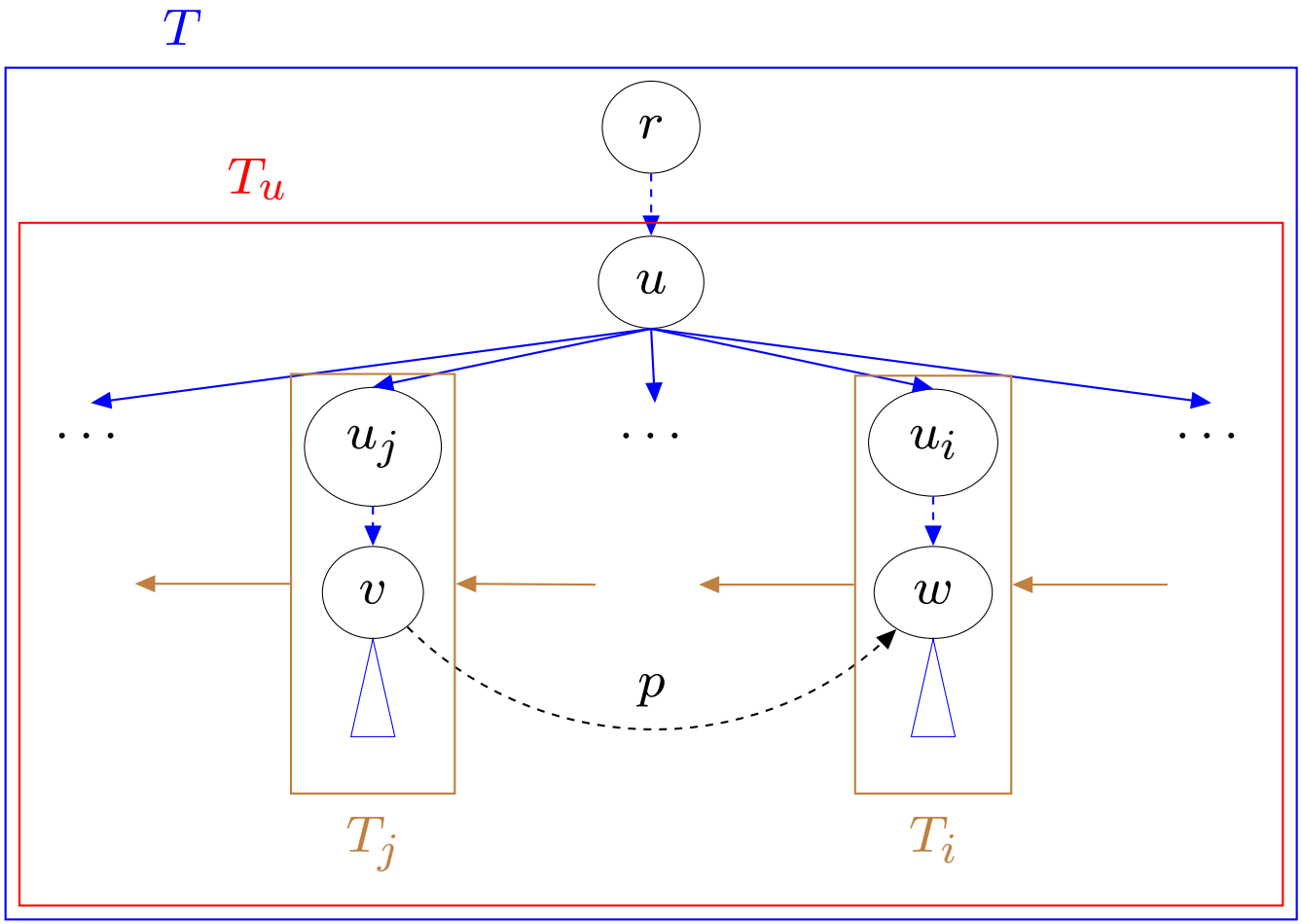
Отметим, что $u \neq w$, так как в этом случае было бы $w \prec v$, что противоречит условию нашего утверждения.

Расположим непосредственных потомков вершины u в дереве T_u в порядке возрастания. В результате у нас получится последовательность вершин u_1, u_2, ..., u_k, являющихся корнями дочерних поддеревьев T_1, T_2, ..., T_k вершины u.

Так как дерево T было получено путём обхода графа G в глубину, то дуга может исходить из вершины дерева T_i и входить в вершину дерева T_j только в том случае, если $j < i$. (Это будет перекрёстная дуга.)

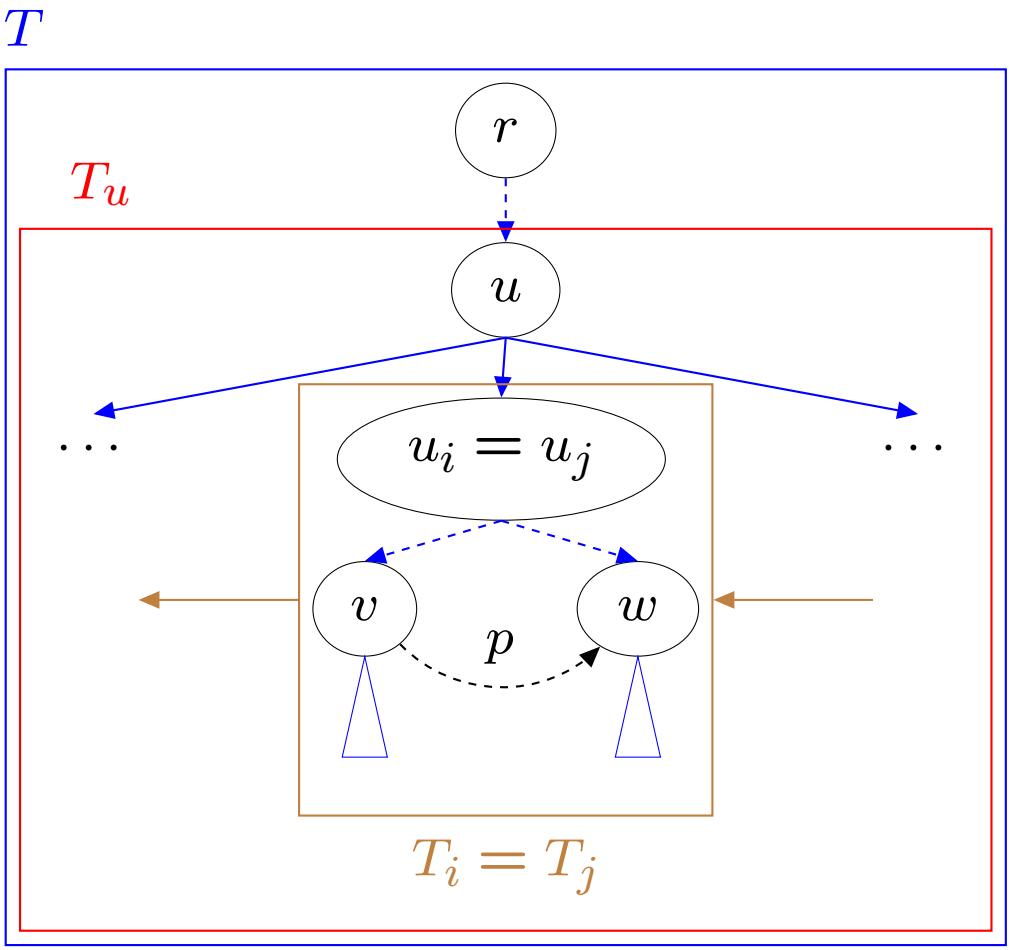


Пусть v принадлежит поддереву T_j, а w принадлежат поддереву T_i, и эти поддеревья различны. Так как $v \prec w$, то $u_j \prec u_i$.



В силу направления поперечных дуг, соединяющих поддеревья вершины u, из вершины v невозможно добраться до вершины w, минуя вершину u.

Если T_i и T_j – одно и то же дочернее поддерево вершины u, то вершина u входит в p в силу минимальности дерева T_u.



◁

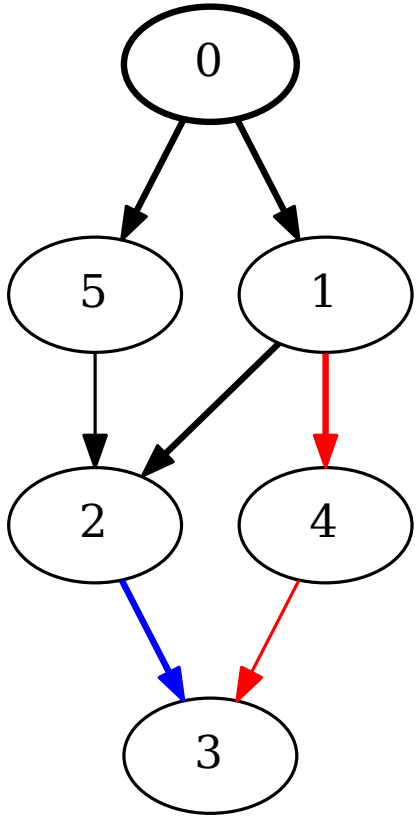
Определение 15.5. Путь $v \to u_1 \to \ldots \to u_i \to \ldots \to u_k \to w$ называется *полудоминаторным*, если $\forall i = \overline{1,k} : \; w \prec u_i$.

Заметим, что согласно определению любой путь единичной длины является полудоминаторным.

Определение 15.6. Вершина v называется *полудоминатором* вершины $w \neq r$, если v – минимальная из вершин, соединённых с w полудоминаторным путём, оканчивающимся на w:
$v = \mathrm{sdom}\,(w) = \min\limits_{\prec} \{u \mid u \to \ldots \to w$ – полудоминаторный путь$\}$.

Понятие полудоминатора важно для алгоритма Ленгауэра–Тарьяна, потому что алгоритм работает в два этапа: сначала для каждой вершины вычисляется её полудоминатор, а потом на основании полудоминатора определяется непосредственный доминатор.

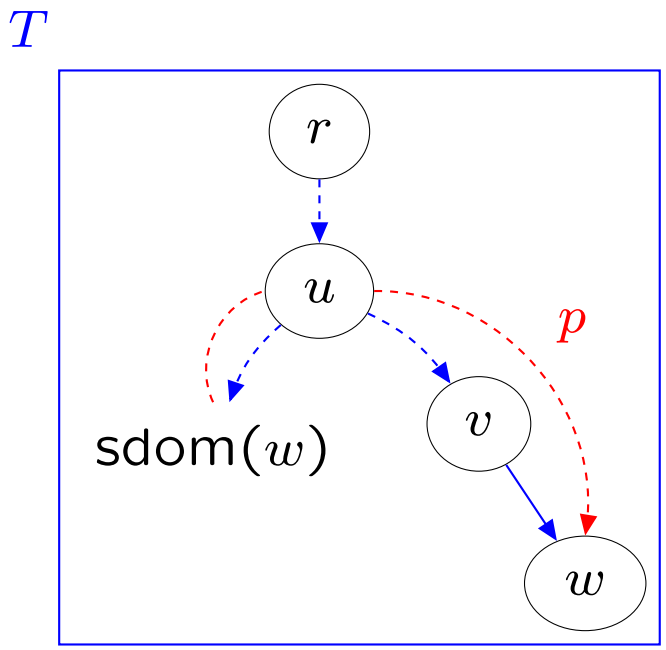
Пример. (Полудоминаторные пути и полудоминаторы.)



К вершине 3 ведут два полудоминаторных пути – красный и синий.
$1 = \mathrm{sdom}(3)$, $0 = \mathrm{idom}(3)$.
(Дуги дерева обхода в глубину – жирные.)

Лемма 15.2. В дереве T существует путь из $\mathrm{sdom}\,(w)$ в $w \neq r$.

▷ Пусть v – непосредственный предок вершины w в дереве T, то есть $v \prec w$. Путь $v \to w$ – полудоминаторный, поэтому по определению полудоминатора $\mathrm{sdom}\,(w) \preceq v$. Отсюда следует, что $\mathrm{sdom}\,(w) \prec w$.



По определению полудоминатора, существует полудоминаторный путь p из $\mathrm{sdom}\,(w)$ в w. Согласно лемме 15.1 путь p содержит общего предка u вершин $\mathrm{sdom}\,(w)$ и w в дереве T.

Так как u – предок w, то $u \prec w$. А так как путь p – полудоминаторный, то все его промежуточные вершины больше w, поэтому u – начальная вершина p, то есть $u = \mathrm{sdom}\,(w)$ – предок w в дереве T.◁

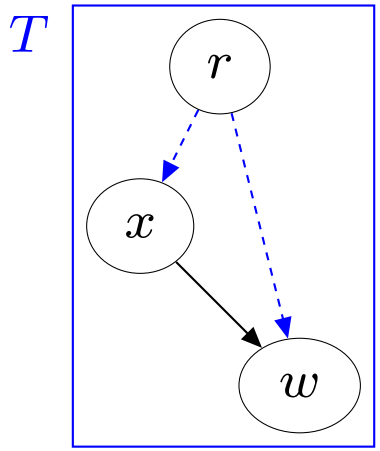
Утверждение 15.1. Пусть $w \neq r$, тогда

$\mathsf{sdom}\,(w) = \min\limits_{\prec}(A \cup B)$, где

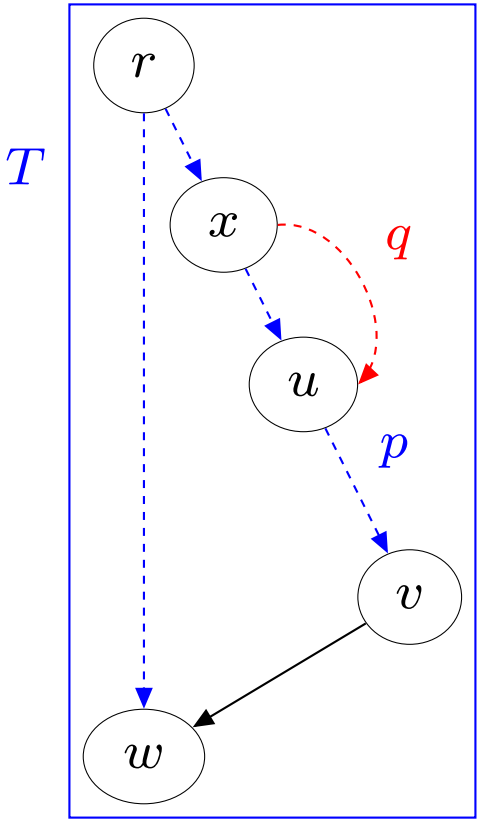
$$\begin{aligned} A &= \{v \mid \langle v, w\rangle \in E\,(G) \text{ и } v \prec w\}, \\ B &= \{\mathsf{sdom}\,(u) \mid w \prec u \text{ и} \\ &\quad \exists\, \langle v, w\rangle \in E\,(G) : \text{ из } u \text{ в } v \text{ в дереве ведёт путь}\}. \end{aligned}$$

▷ Пусть $x = \min\limits_{\prec}(A \cup B)$. Докажем сначала, что $\mathsf{sdom}\,(w) \preceq x$.

Если $x \in A$, т.е. $\langle x, w\rangle \in E\,(G)$ и $x \prec w$, то по определению полудоминатора $\mathsf{sdom}\,(w) \preceq x$.



Если $x \in B$, то $x = \mathrm{sdom}\,(u)$ для некоторой вершины u такой, что $w \prec u$ и существует дуга $\langle v, w \rangle$ такая, что из u в v в дереве T ведёт путь.



По определению полудоминатора существует полудоминаторный путь $q = x \to \ldots \to u$, все промежуточные вершины которого превышают u.

Так как путь $p = u \to \ldots \to v$ – древесный, то все его вершины также превышают или равны u.

В итоге, соединив пути q, p и дугу $\langle v, w \rangle$, мы получим полудоминаторный путь $x \to \ldots \to w$, откуда по определению полудоминатора следует, что $\mathrm{sdom}\,(w) \preceq x$.

Теперь нам осталось доказать, что $x \preceq \mathrm{sdom}\,(w)$.

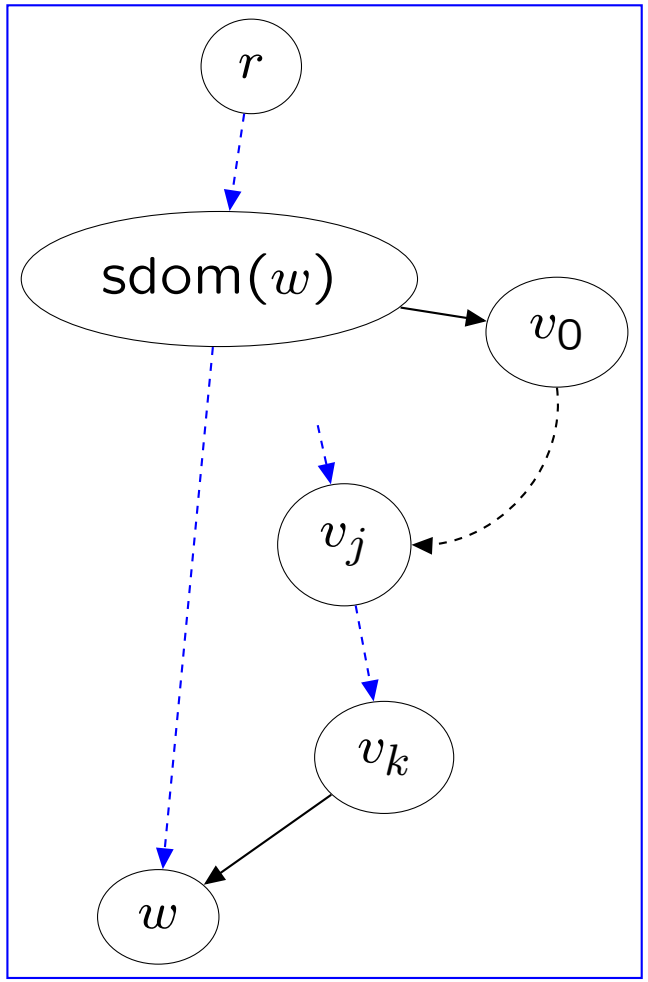
По определению полудоминатора существует полудоминаторный путь $s = \mathrm{sdom}\,(w) \to \ldots \to w$.

Если это путь единичной длины, то в графе существует дуга $\langle \mathrm{sdom}\,(w)\,, w\rangle$, причём, согласно лемме 15.2, $\mathrm{sdom}\,(w) \prec w$.

По условию нашего утверждения $A = \{v \mid \langle v, w\rangle \in E\,(G)$ и $v \prec w\}$. Нетрудно убедиться, что $\mathrm{sdom}\,(w) \in A$.

Учитывая, что $x = \min\limits_{\prec}(A \cup B)$, приходим к выводу, что $x \preceq \mathrm{sdom}\,(w)$.

Теперь рассмотрим случай, когда длина пути s больше 1:
$s = \text{sdom}\,(w) \to v_0 \to \ldots \to v_k \to w$.



Пусть вершина v_j будет первой промежуточной вершиной пути s такой, что существует древесный путь из v_j в v_k. Такая вершина всегда найдётся, так как в крайнем случае $v_j = v_k$.

Предположим, что не все вершины пути s от v_0 до v_{j-1} превышают v_j. Пусть v_i – минимальная среди них. Так как $v_i \prec v_j$, то по лемме 15.1 должен существовать древесный путь $v_i \to \ldots \to v_j$, что противоречит выбору v_j.
Тогда путь $\text{sdom}\,(w) \to \ldots \to v_j$ – полудоминаторный, и по определению полудоминатора $\text{sdom}\left(v_j\right) \preceq \text{sdom}\,(w)$.

Напомним, что

$$B = \{\text{sdom}\,(u) \mid w \prec u \text{ и } \exists\,\langle v, w\rangle \in E\,(G) : \text{ из } u \text{ в } v \text{ в дереве ведёт путь}\}.$$

Нетрудно убедиться, что $\text{sdom}\left(v_j\right) \in B$:
$w \prec v_j$, так как v_j – промежуточная вершина полудоминаторного пути s;
$\exists\,\langle v_k, w\rangle \in E\,(G)$: из v_j в v_k в дереве ведёт путь.

Учитывая, что $x = \min\limits_{\prec}(A \cup B)$, приходим к выводу, что $x \preceq \text{sdom}\left(v_j\right)$. Ну а принимая во внимание, что $\text{sdom}\left(v_j\right) \preceq \text{sdom}\,(w)$, получаем

$x \preceq \text{sdom}\,(w).\triangleleft$

На основании утверждения 15.1 можно построить алгоритм вычисления полудоминаторов.

Итак, $\mathsf{sdom}\,(w) = \min\limits_{\prec}(A \cup B)$, где

$$\begin{aligned} A &= \{v \mid \langle v, w\rangle \in E\,(G) \text{ и } v \prec w\}, \\ B &= \{\mathsf{sdom}\,(u) \mid w \prec u \text{ и} \\ &\quad \exists\, \langle v, w\rangle \in E\,(G) : \text{ из } u \text{ в } v \text{ в дереве ведёт путь}\}. \end{aligned}$$

Обратим внимание на то, что в A входят только вершины, предшествующие w, а в B входят полудоминаторы вершин, которым предшествует w.

Таким образом, для того чтобы вычислить полудоминатор для некоторой вершины w, достаточно знать полудоминаторы вершин, которым предшествует w, а полудоминаторы других вершин могут оставаться невычисленными.

Это наблюдение диктует порядок рассмотрения вершин в алгоритме вычисления полудоминаторов, а именно: вершины должны рассматриваться в порядке убывания.

Пусть в каждой вершине имеется поле $sdom$ для хранения указателя на полудоминатор этой вершины. Если полудоминатор для некоторой вершины w ещё не вычислен, то пусть $w.sdom = w$.

Сначала рассмотрим простой, но неэффективный алгоритм вычисления полудоминаторов:

```
1  Semidominators_naive(in G, in ≺)
2      for each w ∈ V(G):
3          w.sdom ← w
4      for each w ∈ V(G) в порядке убывания по ≺:
5          for each ⟨v, w⟩ ∈ E(G):
6              loop:
7                  if v.sdom ≺ w.sdom:
8                      w.sdom ← v.sdom
9                  if v.sdom = v:
10                     break
11                 v ← v.parent
```

Время работы алгоритма: $O\left(n^2\right)$, где n – количество вершин.

Наивный алгоритм вычисления полудоминаторов неэффективен, главным образом, из-за линейного поиска вершины с минимальным полудоминатором (цикл в строчках 6..11).

Время поиска можно значительно сократить, применив эвристику сжатия пути (аналогично операции Find в лесе непересекающихся множеств).

Добавим в граф вспомогательные размеченные дуги, «срезающие» пути в дереве T. Пусть наличие «срезающей» дуги $\langle v, u \rangle$, помеченной вершиной x, означает, что в дереве T имеется путь $p = u \to \ldots \to x \to \ldots \to v$, на котором вершина x имеет минимальный полудоминатор.

«Срезающие» дуги будут представлены полями $ancestor$ (цель дуги) и $label$ (метка дуги) в вершинах графа.

Рекурсивный алгоритм поиска вершины с минимальным полудоминатором, выполняющий сжатие пути:

```
1  FindMin(in v, in ≺): min
2        SearchAndCut(v, ≺)
3        min ← v.label

5  SearchAndCut(in v, in ≺): root
6        if v.ancestor =NULL:
7              root ← v
8        else:
9              root ←SearchAndCut(v.ancestor, ≺)
10             if v.ancestor.label.sdom ≺ v.label.sdom:
11                   v.label ← v.ancestor.label
12             v.ancestor ← root
```

Нерекурсивная версия алгоритма:

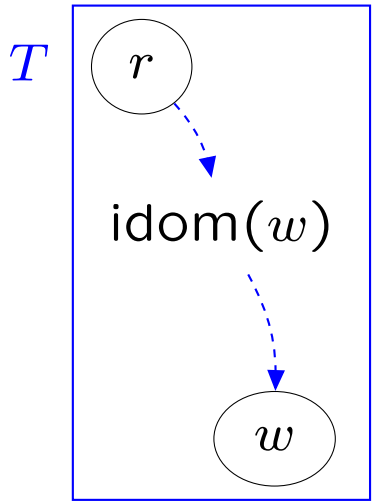
```
1  FindMin(in v, in ≺): min
2        if v.ancestor =NULL:
3              min ← v
4        else:
5              stack ←InitStack(|V(G)|)
6              u ← v
7              while u.ancestor.ancestor ≠NULL:
8                    Push(stack, u)
9                    u ← u.ancestor
10             while not StackEmpty(stack):
11                   v ←Pop(stack)
12                   if v.ancestor.label.sdom ≺ v.label.sdom:
13                         v.label ← v.ancestor.label
14                   v.ancestor ← u.ancestor
15             min ← v.label
```

Алгоритм вычисления полудоминаторов (время – $O(n \lg n)$):

```
1  Semidominators(in G, in ≺)
2      for each w ∈ V(G):
3          w.sdom ← w.label ← w
4          w.ancestor ←NULL
5      for each w ∈ V(G) в порядке убывания по ≺:
6          for each ⟨v, w⟩ ∈ E(G):
7              u ←FindMin(v, ≺)
8              if u.sdom ≺ w.sdom:
9                  w.sdom ← u.sdom
10         w.ancestor ← w.parent
```

Лемма 15.3. В дереве T существует путь из idom (w) в $w \neq r$.

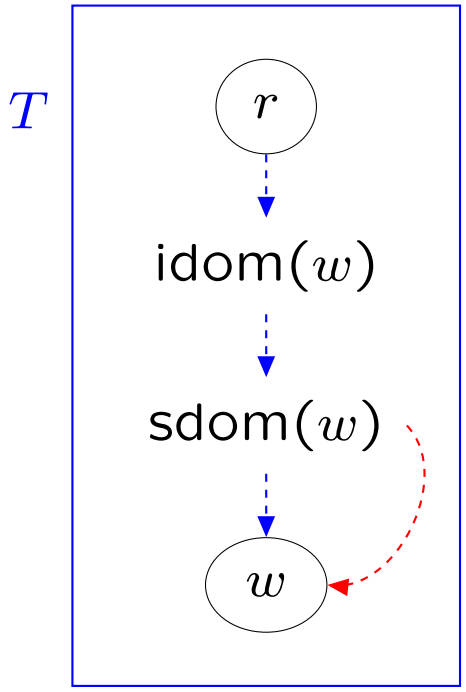
$\triangleright$ Дерево T – подграф G. А согласно определению доминатора, любой путь вида $r \to \ldots \to w$ в графе G проходит через вершину idom (w).



Соответственно, путь из корня дерева в w тоже проходит через idom (w).
$\triangleleft$

Лемма 15.4. В дереве T существует путь из $\mathrm{idom}\,(w)$ в $\mathrm{sdom}\,(w)$ для любой вершины $w \neq r$.

▷ Согласно леммам 15.2 и 15.3, $\mathrm{idom}\,(w)$ и $\mathrm{sdom}\,(w)$ – предки вершины w, значит они расположены на пути из r в w в дереве T.

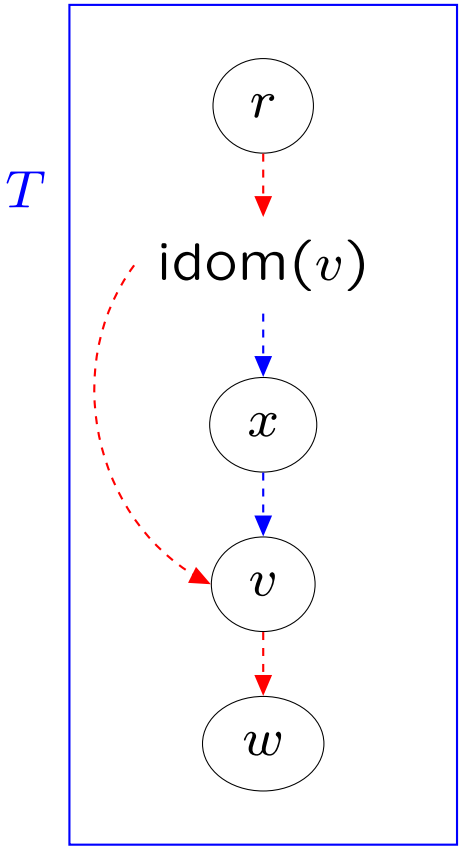


Соединим древесный путь $r \to \ldots \to \mathrm{sdom}\,(w)$ с полудоминаторным путём $\mathrm{sdom}\,(w) \to \ldots \to w$. Так как все промежуточные вершины полудоминарного пути больше w, то они не принадлежат древесному пути из $\mathrm{sdom}\,(w)$ в w.

Поэтому если бы $\mathrm{idom}\,(w)$ принадлежал древесному пути из $\mathrm{sdom}\,(w)$ в w, то наш соединённый путь его бы обходил, что противоречит определению доминатора. ◁

Следствие из леммы 15.4. Для любой вершины $w \neq r$ справедливо: $\mathrm{idom}\,(w) \preceq \mathrm{sdom}\,(w)$.

Лемма 15.5. Пусть в дереве T существует путь из v в w. Тогда в дереве T существует один из двух путей: (a) из v в idom (w), либо (b) из idom (w) в idom (v).



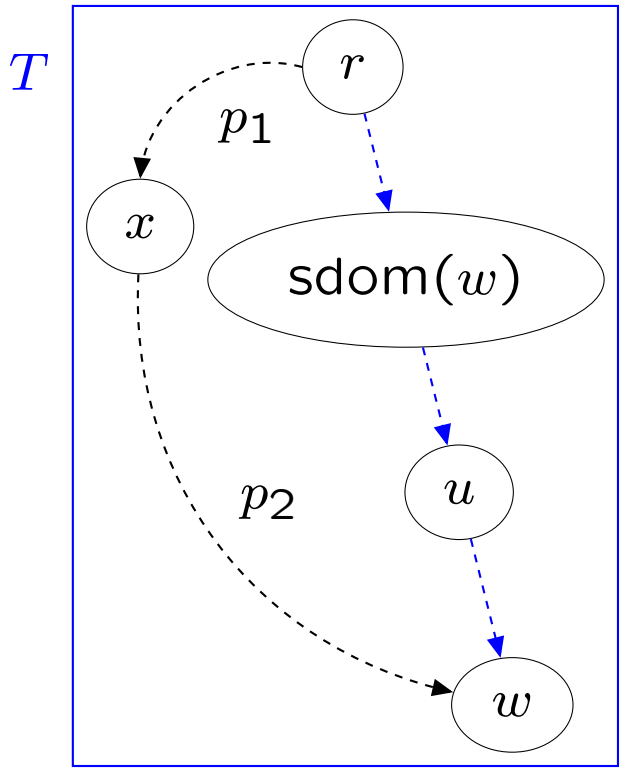
▷ Пусть вершина x находится на пути в дереве из idom (v) к v.

Тогда в графе G существует путь из r к v, обходящий x. Соединив этот путь с древесным путём из v в w, мы получим путь из r к w, обходящий x.

Значит idom (w) не может находиться в дереве между idom (v) и v, то есть idom (w) либо предок idom (v), либо наследник v. ◁

Утверждение 15.2. Пусть $w \neq r$, и среди вершин, расположенных в дереве T на пути из $\mathrm{sdom}\,(w)$ в w и не совпадающих с $\mathrm{sdom}\,(w)$, вершина u имеет минимальный полудоминатор. Тогда если $\mathrm{sdom}\,(u) = \mathrm{sdom}\,(w)$, то $\mathrm{idom}\,(w) = \mathrm{sdom}\,(w)$.

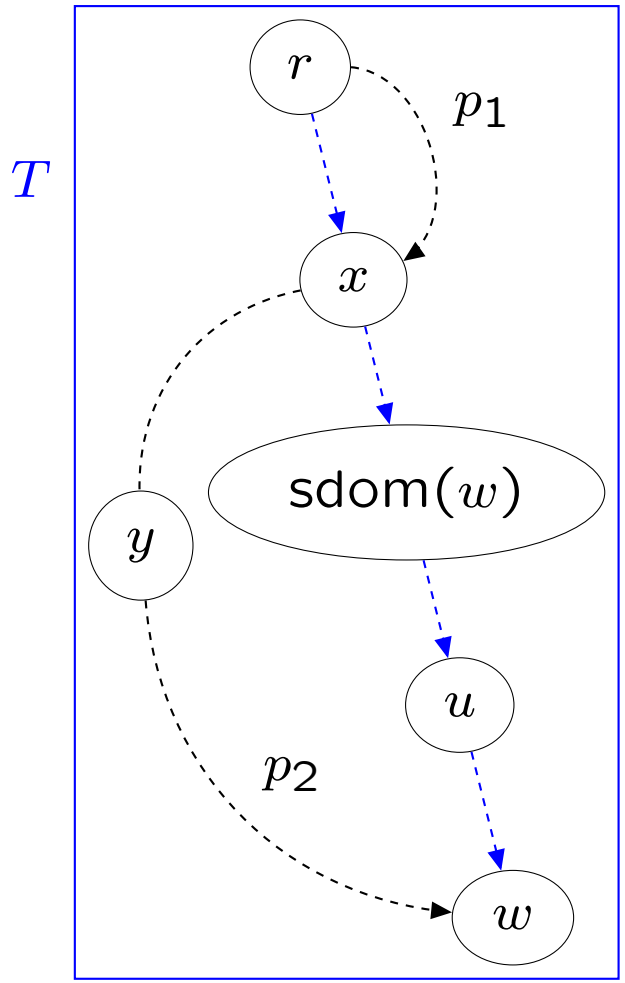
▷ Пусть p – произвольный путь из r к w.



Пусть x – последняя вершина на пути p такая, что $x \preceq \mathrm{sdom}\,(w)$. Она всегда существует, так как в крайнем случае $x = r$. Вершина x разбивает путь p на две части: p_1 и p_2.

По лемме 15.2 $\mathrm{sdom}\,(w) \prec w$, а значит $x \prec w$.

По лемме 15.1 на пути p_2 существует общий предок вершин x и w в дереве T. Так x – минимальная вершина пути p_2, то этот общий предок – вершина x.

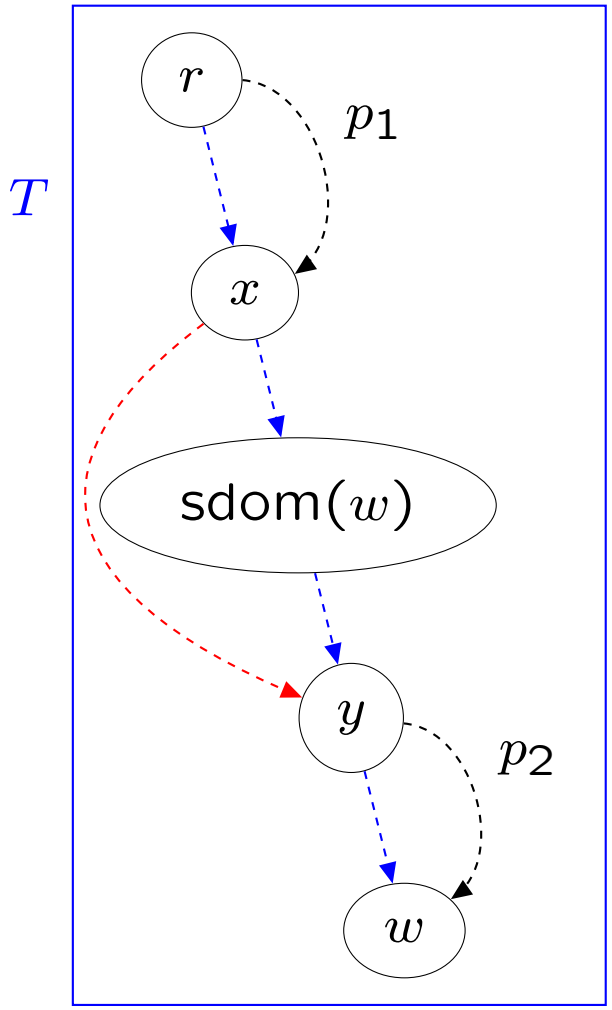


Предположим, что $x \neq \text{sdom}\,(w)$.

Т.к. $x \prec \text{sdom}\,(w)$ в силу сделанного предположения, то путь p_2 – не полудоминаторный.

Значит минимальная промежуточная вершина пути p_2 – назовём её y – предшествует вершине w.

По лемме 15.1 y и w имеют общего предка в дереве T, а т.к. y – минимальная вершина на участке $y \to \ldots \to w$ пути p_2, то этот общий предок – вершина y, причём $\text{sdom}\,(w) \prec y$ в силу выбора вершины x.



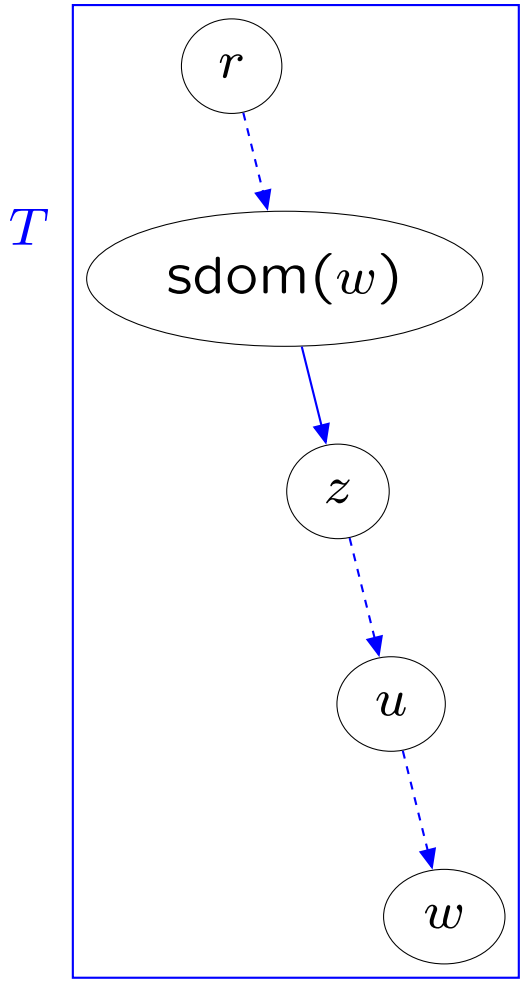
Отрезок $x \to \ldots \to y$ пути p_2 – полудоминаторный путь в силу выбора вершины y. Значит $\text{sdom}\,(y) \preceq x$.

Получается, что $\text{sdom}\,(y) \prec \text{sdom}\,(w)$, что невозможно, т.к. u – вершина с минимальным полудоминатором на древесном пути из $\text{sdom}\,(w)$ в w, и по условию нашего утверждения $\text{sdom}\,(u) = \text{sdom}\,(w)$. Тем самым, мы пришли к противоречию, и $x = \text{sdom}\,(w)$.

Так как мы выбирали путь p произвольно, то любой путь из r в w проходит через $\text{sdom}\,(w)$. Значит $\text{sdom}\,(w)$ доминирует над w, т.е. $\text{sdom}\,(w) \preceq \text{idom}\,(w)$.

По следствию из леммы 15.4, $\text{idom}\,(w) \preceq \text{sdom}\,(w)$, откуда следует, что $\text{idom}\,(w) = \text{sdom}\,(w)$. $\lhd$

Утверждение 15.3. Пусть $w \neq r$, и среди вершин, расположенных в дереве T на пути из $\mathrm{sdom}\,(w)$ в w и не совпадающих с $\mathrm{sdom}\,(w)$, вершина u имеет минимальный полудоминатор.
Тогда $\mathrm{sdom}\,(u) \preceq \mathrm{sdom}\,(w)$ и $\mathrm{idom}\,(w) = \mathrm{idom}\,(u)$.



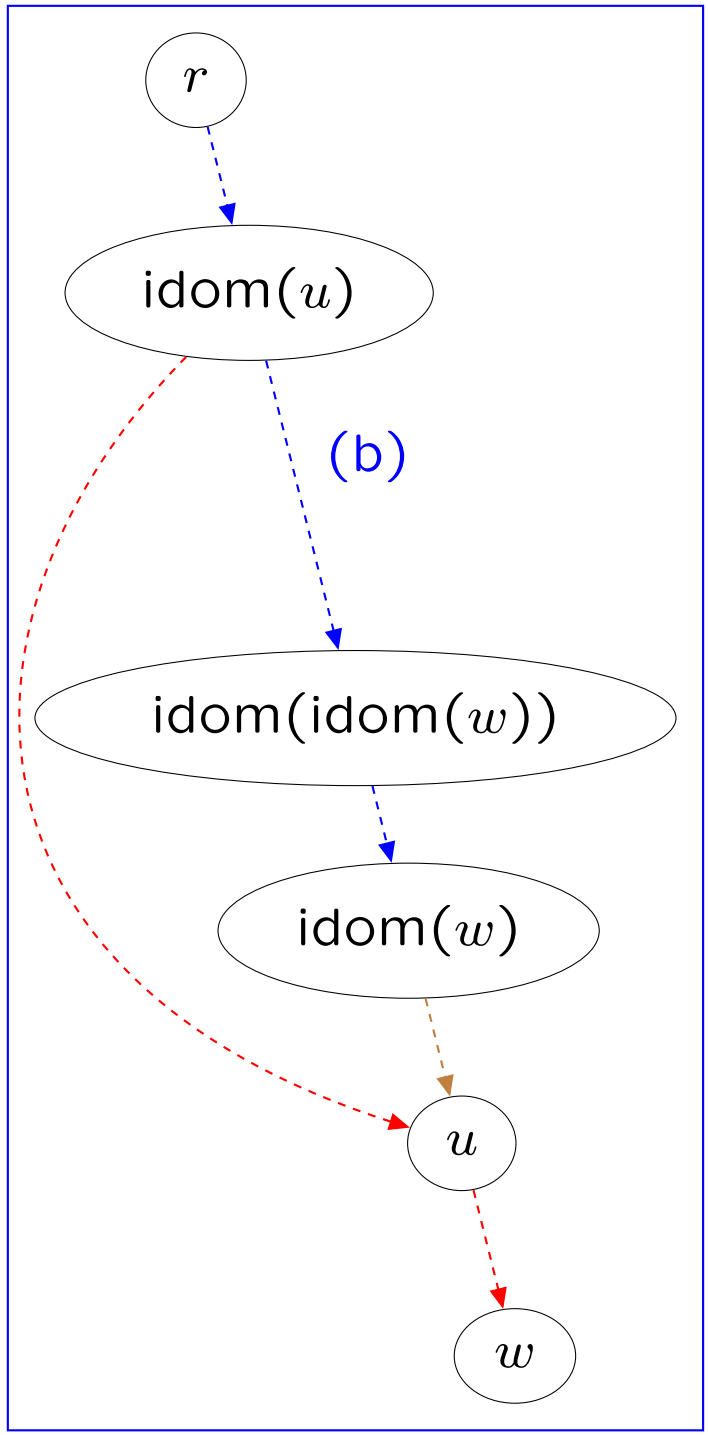
▷ Пусть z – первая после $\mathrm{sdom}\,(w)$ вершина, расположенная в дереве T на пути из $\mathrm{sdom}\,(w)$ в w.

Тогда по условию: $\mathrm{sdom}\,(u) \preceq \mathrm{sdom}\,(z)$.

А так как z непосредственно следует после $\mathrm{sdom}\,(w)$, то $\mathrm{sdom}\,(z) \preceq \mathrm{sdom}\,(w)$.

Объединяя два неравенства, получаем $\mathrm{sdom}\,(u) \preceq \mathrm{sdom}\,(w)$.

Тем самым, мы доказали первую часть утверждения.

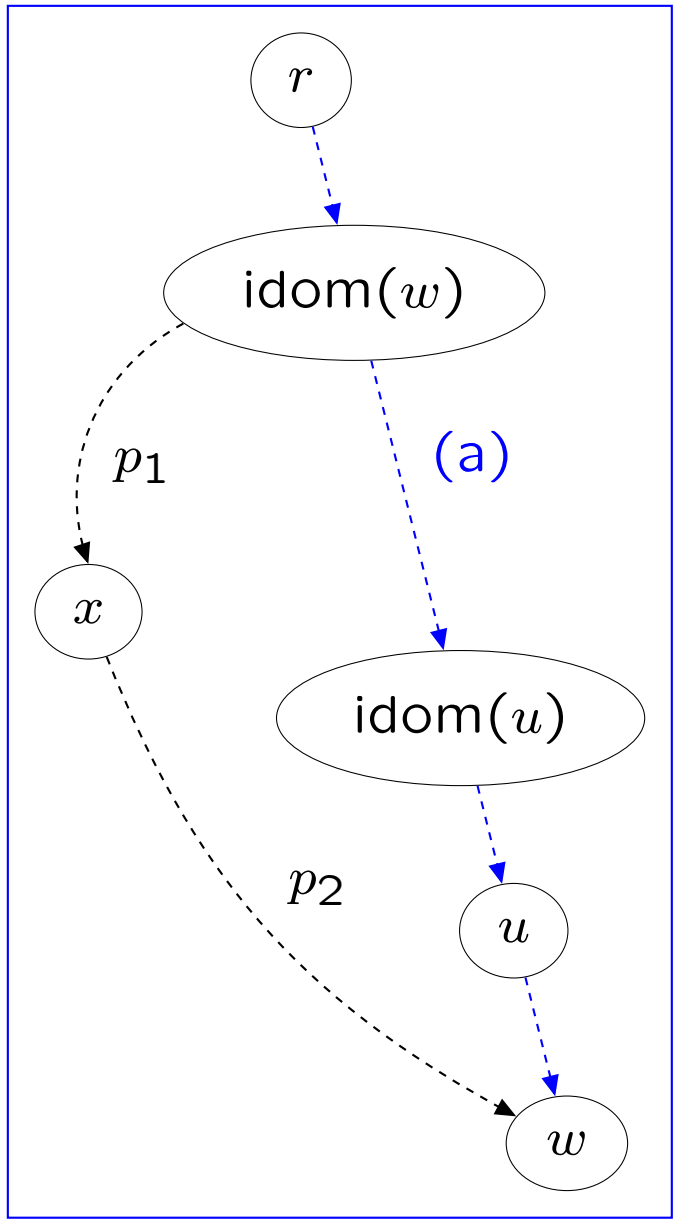


Согласно лемме 15.4, в дереве T существует путь idom $(w) \to \ldots \to$ sdom (w) и, следовательно, путь idom $(w) \to \ldots \to u$.

Тогда по лемме 15.5 в дереве T существует один из двух путей:
(a) из idom (w) в idom (u);
(b) из idom (u) в idom (idom (w)).

Если существует путь (b), то из предположения, что idom $(u) \neq$ idom (w), следует, что можно построить путь из idom (u) в u (и далее в w), обходящий idom (w).

Значит наше предположение неверно, и в случае существования пути (b) вторую часть нашего утверждения можно считать доказанной:
idom $(w) =$ idom (u).

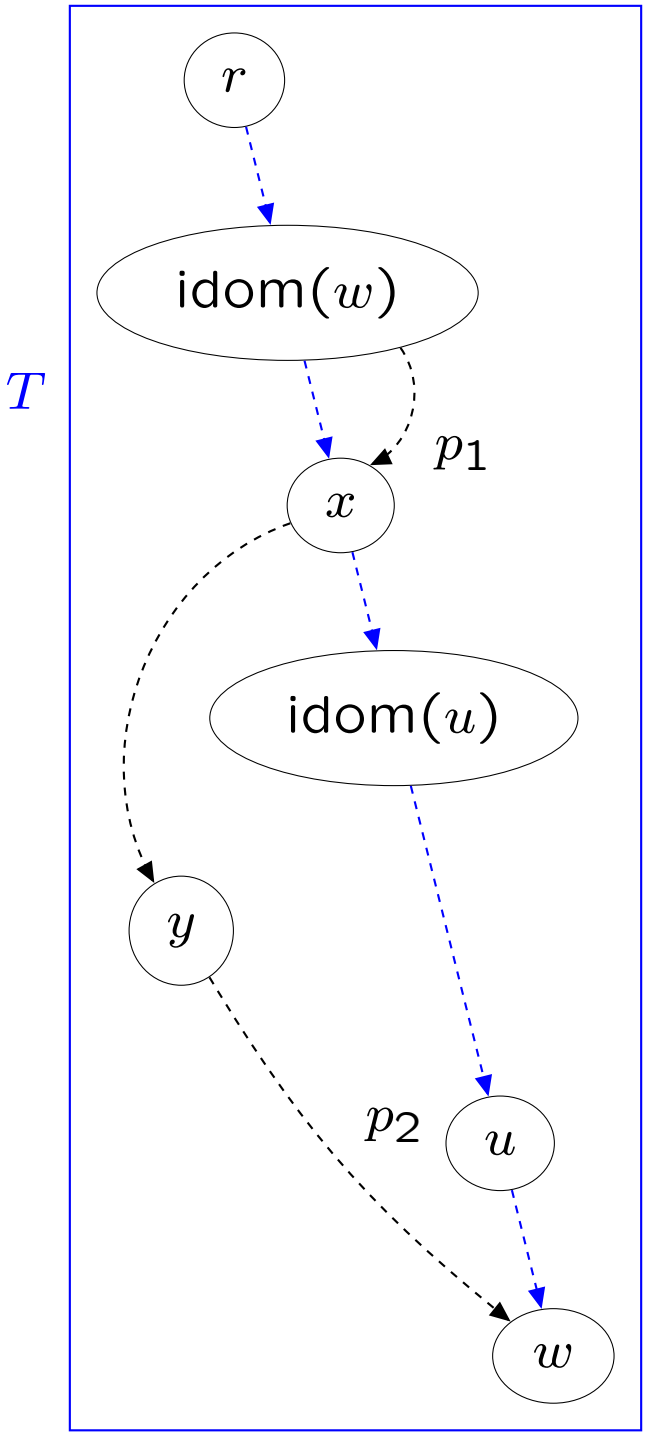


В случае существования в дереве пути (a) из idom (w) в idom (u) для доказательства того, что idom $(w) =$ idom (u), достаточно показать, что idom (u) доминирует над w.

Рассмотрим произвольный путь p из idom (w) в w. Пусть x – последняя вершина на пути p такая, что $x \preceq$ idom (u). Она всегда существует, так как в крайнем случае $x =$ idom (w). Вершина x разбивает путь p на две части: p_1 и p_2.

По лемме 15.3 idom $(u) \prec u$, по условию нашего утверждения $u \preceq w$. Тогда idom $(u) \prec w$, откуда следует, что $x \prec w$.

По лемме 15.1 на пути p_2 существует общий предок вершин x и w в дереве T. Так как x – минимальная вершина пути p_2, то этот общий предок – вершина x.



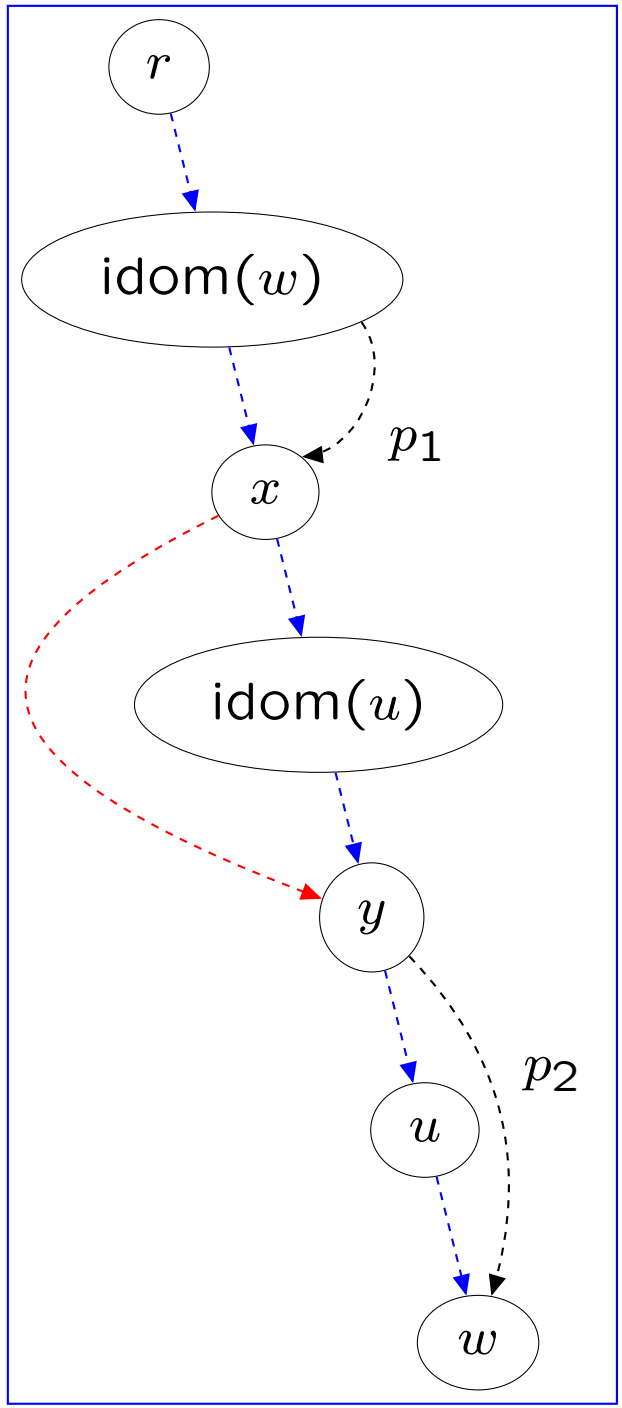
Вершина x не может находиться в дереве T на пути из r в idom (w), т.к. в этом случае мы могли бы объединить древесный путь $r \to \ldots \to x$ с путём p_2, получив тем самым путь из r в w, обходящий idom (w).

Поэтому x находится в дереве T на пути из idom (w) в idom (u).

Предположим, что $x \neq$ idom (u).
Тогда $x \prec$ idom $(u) \preceq$ sdom (u).

Мы доказали, что sdom $(u) \preceq$ sdom (w), поэтому $x \prec$ sdom (w), и путь p_2 – не полудоминаторный.

Пусть y – минимальная промежуточная вершина пути p_2. Тогда $y \prec w$, и по лемме 15.1 y – предок w в дереве. Причём idom $(u) \prec y$ в силу выбора вершины x.



Отрезок $x \to \ldots \to y$ пути p_2 – полудоминаторный путь в силу выбора вершины y. Значит $\mathrm{sdom}\,(y) \preceq x$. А так как $x \prec \mathrm{sdom}\,(u)$, то $\mathrm{sdom}\,(y) \prec \mathrm{sdom}\,(u)$.

Т.к. среди вершин, расположенных в дереве T на пути из $\mathrm{sdom}\,(w)$ в w и не совпадающих с $\mathrm{sdom}\,(w)$, вершина u имеет минимальный полудоминатор, то $y \preceq \mathrm{sdom}\,(w) \prec u$.

Значит y расположена в дереве T на пути $\mathrm{idom}\,(u) \to \ldots \to u$.

Если соединить путь $r \to \ldots \to \mathrm{idom}\,(w)$ с путём p_1, добавить отрезок $x \to \ldots \to y$ пути p_2 и древесный путь из y в u, то получится путь из r в u, не проходящий через $\mathrm{idom}\,(u)$. Тем самым мы пришли к противоречию, и $x = \mathrm{idom}\,(u)$.

Таким образом, $\mathrm{idom}\,(u)$ лежит на любом пути из r к w и, тем самым, доминирует над вершиной w. $\lhd$

Пусть каждая вершина графа имеет поле dom, предназначенное для хранения указателя на непосредственный доминатор вершины.

Модифицируем алгоритм вычисления полудоминаторов, добавив в него код, вычисляющий непосредственные доминаторы.

Нам придётся добавить в каждую вершину поле $bucket$, представляющее подмножество вершин графа.

Поясним смысл поля $bucket$. Пусть имеются некоторые вершины v и w, причём $v = \mathrm{sdom}(w)$. Тогда $w \in v.bucket$, если для вершины w ещё не посчитан непосредственный доминатор.

```
1  Dominators(in G, in ≺)
2      for each w ∈ V(G):
3          w.sdom ← w.label ← w
4          w.ancestor ←NULL
5          w.bucket ← ∅
6      for each w ∈ V(G) \ {r(G)} в порядке убывания по ≺:
7          for each ⟨v, w⟩ ∈ E(G):
8              u ←FindMin(v, ≺)
9              if u.sdom ≺ w.sdom:
10                 w.sdom ← u.sdom
11         w.ancestor ← w.parent
12         w.sdom.bucket ← w.sdom.bucket ∪ {w}
13         for each v ∈ w.parent.bucket:
14             u ←FindMin(v, ≺)
15             v.dom ← if u.sdom = v.sdom: v.sdom else: u
16         w.parent.bucket ← ∅
17     for each w ∈ V(G)\{r(G)} в порядке возрастания по ≺:
18         if w.dom ≠ w.sdom:
19             w.dom ← w.dom.dom
20     r(G).dom ←NULL
```

§16. Постановка задачи поиска кратчайших путей

Пусть дан орграф, каждой дуге которого присвоен вес (целое или вещественное число).

Вес пути в таком орграфе – это сумма весов дуг, входящих в путь.

Если две вершины орграфа соединяются несколькими путями, то *кратчайшим путём* между этими вершинами считается путь с наименьшим весом.

Если задана начальная вершина s, то поиск кратчайших путей в орграфе заключается в вычислении для каждой вершины v двух атрибутов:
$v.dist$ – вес кратчайшего пути из s в v (*оценка кратчайшего пути* для вершины v);
$v.parent$ – вершина, непосредственно предшествующая вершине v на кратчайшем пути из s в v (*родитель* вершины v).

Условимся, что если не существует пути от начальной вершины к вершине v, то $v.dist = \infty$, $v.parent = NULL$.

Будем считать, что перед началом поиска кратчайших путей из начальной вершины s атрибуты вершин установлены следующим образом:
$s.dist = 0$, $s.parent = NULL$,
$v.dist = \infty$, $v.parent = NULL$, если $v \neq s$.

Базовой операцией для алгоритмов поиска кратчайших путей является операция *релаксации дуги*, уточняющая оценку кратчайшего пути для вершины v, в которую входит дуга с весом w, исходящая из вершины u:

```
1 Relax(in u, in v, in w): changed
2         changed ← (u.dist + w < v.dist)
3         if changed:
4                 v.dist ← u.dist + w
5                 v.parent ← u
```

В реализации операции релаксации подразумевается, что $x + \infty = \infty$ и $\infty \not< \infty$. Операция возвращает true, если оценка кратчайшего пути для вершины v была уточнена, и false – в противном случае.

Важным свойством операции релаксации является то, что она не может увеличить оценку кратчайшего пути для вершины v.

Пусть дан орграф $G = \langle V, E \rangle$, начальная вершина $s \in V$, множество вершин $V' \subset V$, для которых посчитаны веса кратчайших путей из s, и некоторая вершина $v \in V \setminus V'$, причём $v.dist = \infty$.

Утверждение 16.1. После релаксации всех дуг, ведущих из вершин V' в v, значение $v.dist$ будет равно весу c кратчайшего пути из s в v в подграфе $G^\star$, порождённом вершинами из $V' \cup \{v\}$.

$\triangleright$ Если ни одной дуги не ведёт из вершин V' в v, то $v.dist$ останется равным ∞. Это правильно, так как в этом случае в орграфе $G^\star$ вершина v не достижима из s, и $c = \infty$. То же самое будет, если все дуги, входящие в v, исходят из вершин, не достижимых из s.

Пусть в вершину v входят дуги, исходящие из вершин, достижимых в орграфе $G^\star$ из s. Предположим, что после релаксации всех этих дуг $v.dist > c$. Так как релаксация не увеличивает $v.dist$, то это условие было верно для каждого вызова операции релаксации.

На кратчайшем пути из s в v вершине v предшествует некоторая вершина u. Поэтому при выполнении релаксации дуги $\langle u, v, w\rangle$ выполнялось неравенство $u.dist + w = c < v.dist$, в результате чего $v.dist$ должно было получить значение c. Значит наше предположение неверно.

Теперь предположим, что после релаксации всех дуг, исходящих из вершин V' и входящих в v, $v.dist < c$. Это значит, что при релаксации некоторой дуги, ведущей из какой-то вершины u в v, значение $u.dist+w$ оказалось меньше c, то есть вес пути $s \to \ldots \to u \to v$ меньше кратчайшего, и наше второе предположение тоже неверно.

Отсюда $v.dist = c.\triangleleft$

§17. Алгоритм Дейкстры

Основная идея алгоритма Дейкстры: на каждом шаге алгоритма выбирается вершина v с минимальной оценкой кратчайшего пути, эта вершина помечается как обработанная (мы к ней больше не вернёмся), и затем производится релаксация всех дуг, исходящих из v:

```
1  Dijkstra(in G, in s)
2      for each v ∈ V(G):
3          v.dist ← ∞
4          v.parent ←NULL
5      s.dist ← 0
6      M ← V(G)
7      while M ≠ ∅:
8          v ←вершина из M с минимальной
9              оценкой кратчайшего пути
10         M ← M \ {v}
11         for each ⟨v, u, w⟩ ∈ E(G):
12             Relax(v, u, w)
```

Алгоритм Дейкстры работает в случае, когда веса дуг – неотрицательные.

В случае линейного поиска вершины с минимальной оценкой кратчайшего пути сложность алгоритма Дейкстры: $O\left(n^2 + m\right)$, где n – количество вершин, m – количество дуг.

Утверждение 17.1. Вес кратчайшего пути от вершины s до вершины v, которая выбирается на каждой итерации внешнего цикла алгоритма Дейкстры, равен оценке кратчайшего пути для вершины v.

▷ Доказательство утверждения проводится индукцией по номеру итерации.

На первой итерации, очевидно, выбирается вершина s с оценкой пути 0, которая равна весу кратчайшего пути из s в s.

Пусть на i-той итерации выбрана вершина v. При этом веса кратчайших путей для $(i-1)$ вершин, удалённых из множества M на предыдущих итерациях, уже посчитаны.

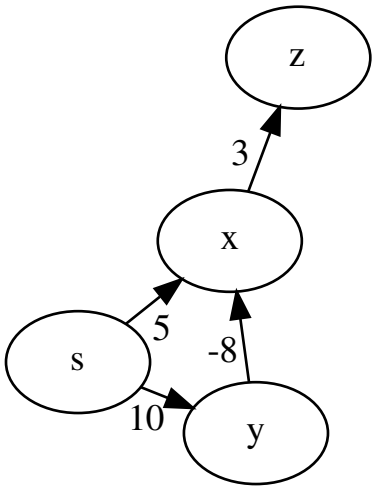
Так как на предыдущих итерациях были релаксированы все дуги, исходящие из вершин, принадлежащих $V(G) \setminus M$, то, согласно утверждению 16.1, $v.dist$ равно весу кратчайшего пути из s в v в подграфе $G^\star$, образованном вершинами $(V(G) \setminus M) \cup \{v\}$.

Предположим, что в графе G имеется путь из s в v, вес которого $c < v.dist$. Тогда этот путь выходит за пределы подграфа $G^\star$, то есть выглядит как $s \to \ldots \to x \to y \to \ldots \to v$, где $x \in V(G) \setminus M$, а $y \in M$.

Вес пути $s \to \ldots \to x \to y$ не меньше, чем $y.dist$, так как, согласно утверждению 16.1, $y.dist$ равно весу кратчайшего пути из s в y в подграфе $G^{\star\star}$, образованном вершинами $(V(G) \setminus M) \cup \{y\}$, а путь $s \to \ldots \to x \to y$ не выходит за пределы $G^{\star\star}$. Тогда вес пути $s \to \ldots \to x \to y \to \ldots \to v$ тоже не меньше, чем $y.dist$, то есть $y.dist \leq c$.

Вершина v имеет минимальную оценку кратчайшего пути среди всех вершин из M, значит $v.dist \leq y.dist \leq c$.$\lhd$

Пример. (алгоритм Дейкстры не работает для графов с отрицательными весами)



В результате работы алгоритма Дейкстры $z.dist = 8$, в то время как путь $s \to y \to x \to z$ имеет вес 5.

Алгоритм Дейкстры целесообразно реализовывать через очередь с приоритетами:

```
1  Dijkstra(in G, in s)
2      InitPriorityQueue(out q, V(G))
3      for each v ∈ V(G):
4          if v = s:
5              v.dist ← 0
6          else:
7              v.dist ← ∞
8          v.parent ←NULL
9          QueueInsert(q, v)
10     while not QueueEmpty(q):
11         v ←ExtractMin(q)
12         v.index ← −1
13         for each ⟨v, u, w⟩ ∈ E(G):
14             if u.index ≠ −1 and Relax(v, u, w):
15                 DecreaseKey(q, u.index, u.dist)
```

В случае использования очереди с приоритетами, реализованной на базе двоичной пирамиды, сложность алгоритма Дейкстры: $O((n+m)\cdot \lg n)$, где n – количество вершин, m – количество дуг.

§18. Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда может работать в случае, когда веса части дуг орграфа – отрицательные.

В основе алгоритма лежит следующее соображение: если в орграфе есть кратчайший путь $s \to x \to y \to \ldots \to u \to v$, то выполняя релаксацию дуг $\langle s, x\rangle$, $\langle x, y\rangle$, $\ldots$, $\langle u, v\rangle$ в той последовательности, в какой они входят в путь, мы получим правильно установленные атрибуты $dist$ и $parent$ для всех вершин кратчайшего пути.

Кратчайший путь в графе не может содержать больше, чем $|V| - 1$ вершин, а операция релаксации не может увеличивать оценку $dist$, поэтому, чтобы обеспечить нахождение всех кратчайших путей, можно $|V| - 1$ раз запустить релаксацию всех дуг графа.

Алгоритм Беллмана-Форда:

```
1  BellmanFord(in G, in s)
2        for each v ∈ V(G):
3              v.dist ← ∞
4              v.parent ←NULL
5        s.dist ← 0
6        i ← 1
7        while i < |V(G)|:
8              for each ⟨u, v, w⟩ ∈ E(G):
9                    Relax(u, v, w)
10             i ← i + 1
```

Время выполнения алгоритма Беллмана-Форда: $O(nm)$, где n – количество вершин, m – количество дуг.

Если граф содержит циклы с отрицательным весом, то, очевидно, вес некоторых путей в графе может становиться сколь угодно мал. Чтобы обнаружить наличие таких циклов, достаточно запустить операцию релаксации ещё один раз и посмотреть, изменились ли оценки кратчайшего пути в вершинах графа.

```
1  BellmanFord(in G, in s): negative_cycles
2      for each v ∈ V(G):
3          v.dist ← ∞
4          v.parent ←NULL
5      s.dist ← 0
6      i ← 1
7      while i < |V(G)|:
8          for each ⟨u, v, w⟩ ∈ E(G):
9              Relax(u, v, w)
10         i ← i + 1
11     for each ⟨u, v, w⟩ ∈ E(G):
12         if Relax(u, v, w):
13             return true
14     return false
```

§19. Топологическая сортировка ациклического орграфа

Утверждение 19.1. Орграф не содержит циклов тогда и только тогда, когда в нём нет обратных дуг.

▷Необходимость. Пусть дан ациклический ограф. Докажем, что при любом его обходе в глубину не будет ни одной обратной дуги.

Предположим, что какой-то обход в глубину выявил обратную дугу $\langle u, v\rangle$. Обратная дуга соединяет вершину u с её предком v в дереве обхода в глубину. Значит в дереве имеется путь $v \mapsto u$. Обратная дуга $\langle u, v\rangle$ замыкает этот путь в цикл.

Достаточность. Пусть ни один обход графа в глубину не выявляет ни одной обратной дуги.

Предположим, что граф содержит цикл c, а вершина v, принадлежащая циклу c, имеет минимальное время захода среди всех вершин цикла. Очевидно, что все остальные вершины цикла являются потомками v в дереве обхода в глубину, включая вершину u, непосредственно предшествующую вершине v в цикле. Поэтому дуга $\langle u, v\rangle$ – обратная.◁

Топологический порядок на множестве вершин ациклического орграфа $G = \langle V, E \rangle$ – это такое бинарное отношение $\overset{top}{\prec}$, что

$$\forall u, v \in V : \; u \overset{top}{\prec} v \iff \exists \text{путь } u \mapsto v.$$

Отношение $\overset{top}{\prec}$ является отношением частичного порядка, т.к. обладает следующими свойствами:

- рефлексивность: $\forall u : \; u \overset{top}{\prec} u$;
- антисимметричность: $\forall u, v : \; u \overset{top}{\prec} v \; \wedge v \overset{top}{\prec} u \; \Rightarrow \; u = v$;
- транзитивность: $\forall u, v, w : \; u \overset{top}{\prec} v \; \wedge v \overset{top}{\prec} w \; \Rightarrow \; u \overset{top}{\prec} w$.

Напомним, что в процессе обхода в глубину каждой вершине x графа можно присвоить время захода $x.T1$ и время выхода $x.T2$.

Утверждение 19.2. Если $u \overset{top}{\prec} v$, где u и v – вершины ациклического орграфа G, то для любого обхода в глубину $v.T2 < u.T2$.

$\triangleright$По определению топологического порядка, существует путь $u \mapsto v$. Согласно утверждению 19.1, при любом обходе в глубину этот путь не содержит обратных дуг.

Рассмотрим любую дугу $\langle x, y\rangle$, принадлежащую пути $u \mapsto v$.

Если $\langle x, y\rangle$ – древесная или прямая, то $x.T1 < y.T1 < y.T2 < x.T2$.

Если $\langle x, y\rangle$ – поперечная, то $y.T2 < x.T1$, и, следовательно, $y.T2 < x.T2$.

Исходя из этих соотношений, индукцией по номеру дуги на пути $u \mapsto v$ можно показать, что $v.T2 < u.T2$. $\triangleleft$

Топологическая сортировка ациклического орграфа – это формирование последовательности $\{u_i\}$, составленной из всех вершин графа таким образом, что

$$\forall u_i, u_j : \; u_i \overset{top}{\prec} u_j \; \Rightarrow \; i < j.$$

Алгоритм топологической сортировки, предложенный Тарьяном, составлен на основе следующего соображения.

Согласно утверждению 19.2:

$$\forall u_i, u_j : \; u_i \overset{top}{\prec} u_j \; \Rightarrow \; u_j.T2 < u_i.T2.$$

Поэтому для получения топологической сортировки достаточно выполнить обход орграфа в глубину и расположить его вершины в порядке убывания времени выхода (другими словами, достаточно построить *обратный постпорядок*):

$$\forall u_i, u_j : \; i < j \; \Leftrightarrow \; u_j.T2 < u_i.T2.$$

§20. Транзитивное замыкание орграфа

Пусть дана пара вершин u и v орграфа G и требуется выяснить, существует ли путь $u \mapsto v$. Эта задача называется *запросом достижимости*.

Самое простое решение запроса достижимости заключается в запуске обхода орграфа из вершины u. Если в процессе обхода будет достигнута вершина v, значит путь существует. В противном случае, путь не существует.

Обход орграфа выполняется за время $O(n + m)$, где n и m – количества вершин и дуг орграфа, соответственно. Поэтому в случае, когда нужно выполнить большое количество запросов достижимости, такой подход оказывается неэффективным.

Существует ряд эффективных алгоритмов вычисления запросов достижимости. Все эти алгоритмы работают на ациклических орграфах. Если граф, на котором требуется выполнять запросы достижимости, содержит циклы, сначала строится его конденсация.

Конденсация орграфа G – это орграф, множество вершин которого состоит из компонент сильной связности орграфа G, а множество дуг содержит дугу $\langle u, v\rangle$ тогда и только тогда, когда в графе G имеется дуга, исходящая из вершины компоненты u и заходящая в вершину компоненты v.

Утверждение 20.1. Конденсация орграфа не содержит циклов.

▷Предположим, что конденсация орграфа G содержит цикл c.

Возьмём две разные компоненты сильной связности, принадлежащие этому циклу, и выберем в каждой из них произвольную вершину. Обозначим эти вершины как u и v.

Легко показать, что из существования цикла c следует существование в графе G двух путей: $u \mapsto v$ и $v \mapsto u$, т.е. вершины u и v взаимно достижимы и должны принадлежать одной компоненте сильной связности графа G.◁

Итак, для любого орграфа G можно построить его конденсацию C.

Пусть вершины u и v принадлежат компонентам сильной связности c_u и c_v, соответственно. Очевидно, что путь $u \mapsto v$ существует в орграфе G тогда и только тогда, когда в его конденсации C существует путь $c_u \mapsto c_v$.

Это соображение позволяет нам в дальнейшем рассматривать вычисление запросов достижимости только для ациклических орграфов.

Вычислять запросы достижимости за константное время можно, построив так называемое транзитивное замыкание органа.

Если на множестве вершин ациклического орграфа $G = \langle V, E \rangle$ построено отношение частичного порядка $\overset{top}{\prec}$, то его *транзитивным замыканием* будет орграф $TC(G) = \left\langle V, \overset{top}{\prec} \right\rangle$.

Очевидно, что в орграфе G существует путь $u \to v$ тогда и только тогда, когда в орграфе $TC(G)$ содержится дуга $\langle u, v \rangle$.

Размер транзитивного замыкания в памяти равен $O\left(n^2\right)$.

Алгоритм построения транзитивного замыкания перебирает вершины ациклического оргафа в обратном топологическом порядке и для каждой вершины строит множество исходящих из неё дуг в транзитивном замыкании.

При построении множества исходящих дуг используется тот факт, что дуги транзитивного замыкания для вершин, непосредственно достижимых из данной вершины, уже построены.

```
1 TransitiveClosure(in G): TC
2       V(TC) ← V(G)
3       E(TC) ← ∅
4       for each u ∈ V(G) в обр. топологическом порядке:
5           E(TC) ← E(TC) ∪ {⟨u, u⟩}
6           for each ⟨u, v⟩ ∈ E(G):
7               for each ⟨v, w⟩ ∈ E(TC):
8                   E(TC) ← E(TC) ∪ {⟨u, w⟩}
```

Время работы алгоритма – $O(nm)$, так как для каждой вершины перебирается не более m дуг.

§21. Интервальное кодирование транзитивного замыкания дерева

Если орграф является деревом, то можно применить эффективную схему представления его транзитивного замыкания, требующую $O(n)$ памяти.

Пусть выполнен обход дерева в глубину от корня, и в каждой вершине дерева записано её время выхода $T2$.

Из любой вершины дерева u достижимы только вершины, принадлежащие растущему из неё поддереву. При этом у всех этих вершин время выхода $T2$ меньше, чем $u.T2$.

Пусть у каждой вершины имеется поле $index$, равное минимальному времени выхода для вершин растущего из неё поддерева.

Тогда $u \mapsto v \;\Leftrightarrow\; u.index \leq v.T2 \leq u.T2$.

Алгоритм построения интервалов, кодирующих транзитивное замыкание ориентированного дерева:

```
1 TreeClosure(in/out T)
2       time ← 0
3       for each u ∈ V(T) в обр. топологическом порядке:
4           u.T2 ← time
5           if u – листовая вершина:
6               u.index ← time
7           else:
8               u.index ← min{v.index | ⟨u, v⟩ ∈ E(T)}
9           time ← time + 1
```

Алгоритм выполнения запроса достижимости:

```
1 TreeQuery(in T, in u, in v): verdict
2       verdict ← u.index ≤ v.T2 ≤ u.T2
```

Утверждение 21.1. Любые два интервала, кодирующие транзитивное замыкание ориентированного дерева, либо не пересекаются, либо один из них вложен в другой.

▷Предположим, что интервалы, вычисленные для двух вершин дерева u и v пересекаются, но при этом ни один из них не вложен в другой, т.е.

$$v.index < u.index \leq v.T2 < u.T2.$$

Так как $v.T2$ попадает в интервал, соответствующий вершине u, то $u \mapsto v$.

Обозначим как w вершину с временем выхода $v.index$. Тогда $v \mapsto w$.

Получается, что $u \mapsto w$, но при этом $w.T2$ не попадает в интервал, соответствующий вершине u. ◁

§22. Интервальное кодирование транзитивного замыкания ациклического орграфа

Пусть дан ациклический орграф G. Построим для этого орграфа остовный лес. Это можно сделать простым обходом в глубину, а можно воспользоваться способом, который будет рассмотрен в дальнейшем.

Пусть транзитивное замыкание для деревьев остовного леса закодировано интервалами в каждой вершине леса. Для построения этих интервалов можно воспользоваться алгоритмом TreeClosure, т.к. он, очевидно, годится не только для ориентированного дерева, но и для ориентированного леса.

Ясно, что интервалы в вершинах остовного леса не годятся для вычисления запросов достижимости в орграфе G, так как они построены без учёта дуг орграфа G, не попавших в остовный лес.

Запишем алгоритм, который строит в каждой вершине u орграфа G множество интервалов $u.ranges$, состоящее из интервалов вершин, достижимых из u (включая и интервал, которым помечена сама вершина u):

$$u.ranges = \{\langle v.index, v.T2\rangle \mid u \mapsto v\},$$

и выбрасывает из этого множества все интервалы, которые вкладываются в другие интервалы множества:

```
1 DagClosure(in/out G)
2     for each u ∈ V(G) в обр. топологическом порядке:
3         u.ranges ← {⟨u.index, u.T2⟩}
4         for each ⟨u, v⟩ ∈ E(G):
5             u.ranges ← MergeRanges(u.ranges, v.ranges)
```

Вспомогательный алгоритм MergeRanges объединяет два множества интервалов, не включая в результат интервалы, вкладывающиеся в другие интервалы.

Согласно утверждению 21.1, интервалы в остовном лесу либо не пересекаются, либо вкладываются. Так как множество интервалов, соответствующее вершине орграфа, избавлено от интервалов, вкладывающихся в другие интервалы множества, то получается, что все интервалы не пересекаются.

Эффективным представлением множества интервалов является последовательность, в которой интервалы отсортированы по возрастанию, т.е. для любых двух подряд идущих интервалов $a = \langle a_l, a_r \rangle$ и $b = \langle b_l, b_r \rangle$ справедливо, что $a_r < b_l$.

Такое представление позволяет реализовать алгоритм MergeRanges по схеме, похожей на алгоритм слияния двух последовательностей в алгоритме сортировки слиянием.

```
1  MergeRanges(in a, in b): res
2      res ← пустая последовательность
3      i ← 0;   j ← 0
4      while i < Len(a) or j < Len(b):
5          if j = Len(b):
6              добавить a[i] в конец res;   i ← i + 1
7          else if i = Len(a):
8              добавить b[j] в конец res;   j ← j + 1
9          else let a[i] = ⟨a_l, a_r⟩, b[j] = ⟨b_l, b_r⟩:
10             if a_r < b_l:
11                 добавить a[i] в конец res;   i ← i + 1
12             else if b_r < a_l:
13                 добавить b[j] в конец res;   j ← j + 1
14             else if b_l ≤ a_l and a_r ≤ b_r:
15                 i ← i + 1
16             else:
17                 j ← j + 1
```

Очевидно, что вершина v орграфа G достижима из вершины u тогда и только тогда, когда существует интервал $\langle l, r\rangle \in u.ranges$ такой, что $l \le v.T2 \le r$.

Так как интервалы в последовательности $u.ranges$ отсортированы, то поиск интервала, в котором лежит $v.T2$, можно организовать методом деления пополам:

```
1  DagQuery(in G, in u, in v): verdict
2          verdict ← false
3          left ← 0;    right ← Len(u.ranges)
4          while left < right:
5                  m ← left + (right − left)/2
6                  let u.ranges[m] = ⟨l, r⟩:
7                          if v.T2 < l:
8                                  right ← m
9                          else if r < v.T2:
10                                 left ← m + 1
11                         else:
12                                 verdict ← true
13                                 break
```

Теперь представим способ построения остовного леса, оптимальный с точки зрения минимизации суммарного количества интервалов в вершинах орграфа G.

Очевидно, что если в вершину v входят две дуги – $\langle u_1, v\rangle$ и $\langle u_2, v\rangle$, причём дуга $\langle u_1, v\rangle$ принадлежит лесу, то в множестве интервалов для любого предка u_1 в лесу найдётся интервал, поглощающий интервал $\langle v.index, v.T2\rangle$. Что же касается предков вершины u_2, то интервал $\langle v.index, v.T2\rangle$ будет поглощаться только для тех из них, кто одновременно является предком u_1.

```
1  TreeCover(in G): T
2      for each v ∈ V(G):
3          v.pred ← 1

5      V(T) ← V(G);    E(T) ← ∅
6      for each v ∈ V(G) в топологическом порядке:
7          for each w : v ↦ w :
8              w.pred ← w.pred + 1
9          max ←NULL
10         for each ⟨u, v⟩ ∈ E(G):
11             if max =NULL or max.pred < u.pred:
12                 max ← u
13         if max ≠NULL:
14             E(T) ← E(T) ∪ {⟨max, v⟩}
```