Введение в дискретную математику и математическую логику

- Лекция №5

Остовные деревья

- Апанович Зинаида Владимировна

Введение

Рассмотрим систему дорог в Новосибирской области, представленную простым графом, показанным ниже.

Единственный способ сохранить дороги доступными зимой — это их частая расчистка.

При этом желательно расчищать как можно меньше дорог, НО так чтобы всегда были расчищены дороги, соединяющие любые два города.

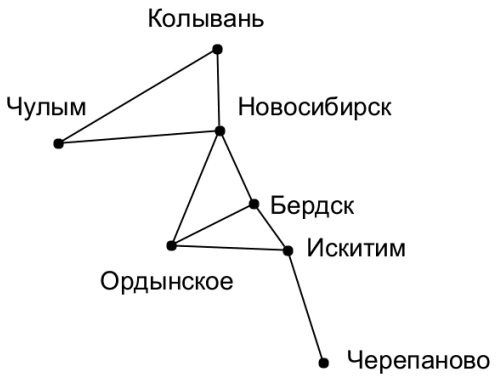
Как это можно сделать?

Чтобы обеспечить проезд между любыми двумя городами, необходимо расчистить не менее 6 дорог. На рисунке 1(б) показано одно из таких множеств дорог.

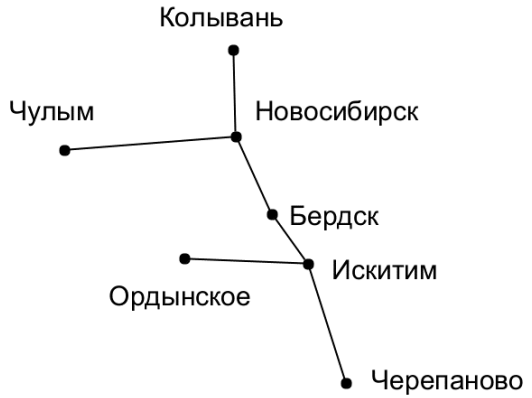
Обратите внимание, что подграф , представляющий эти дороги, является деревом, потому что он связен и содержит 6 вершин и 5 ребер.

Эта задача была решена с помощью связного подграфа с минимальным числом ребер, содержащим все вершины исходного простого графа.

Такой граф должен быть деревом.



(a) Система дорог.



(b) Дерево дорог для расчистки.

Остовное дерево графа *G*

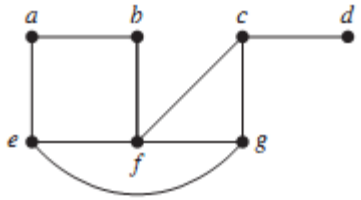
ОПРЕДЕЛЕНИЕ 1. Пусть *G* — простой граф.

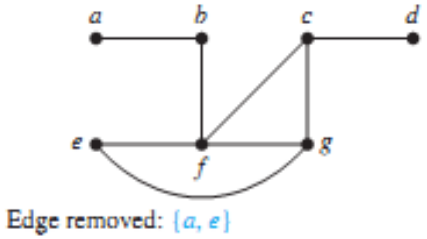
Остовное дерево графа *G* — это подграф графа *G*, представляющий собой дерево, содержащее все вершины графа *G.*

Простой граф, имеющий остовное дерево должен быть связным, поскольку в остовном дереве существует путь между любыми двумя вершинами.

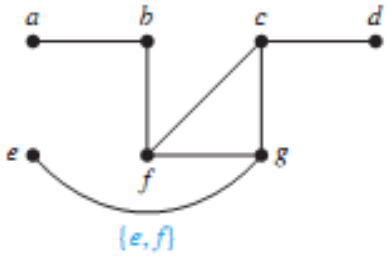
Обратное также верно, то есть, каждый связанный простой граф имеет остовное дерево.

Остовное дерево графа *G*

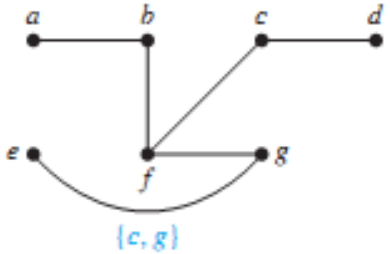




(a)



(b)



(c)

ПРИМЕР. Найдите остовное дерево простого графа *G.*
Решение : Граф *G* связен, но не является деревом, поскольку содержит простые циклы.
Удалим ребро { *a, e* }.
Это устраняет 1 простой цикл,

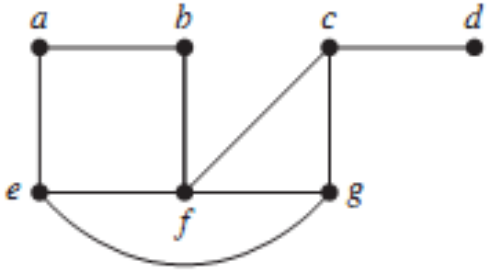
Затем удалим ребро { *e, f* } , чтобы исключить второй простой цикл.

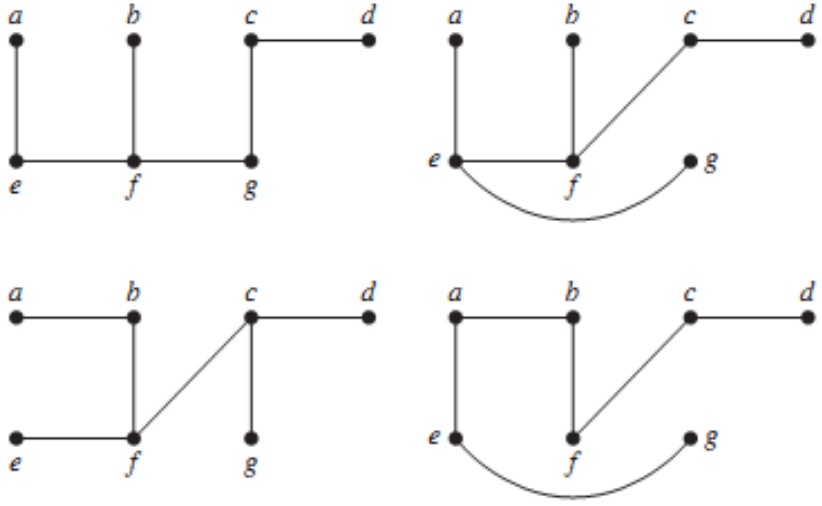
Наконец, удалим ребро { *c, g* } , чтобы получить простой граф без простых циклов.
Этот подграф является остовным деревом, поскольку это дерево, содержащее каждую вершину графа *G.*

Другие остовные деревья *G*

Дерево, показанное в примере 1, не является единственным остовным деревом графа *G.*

Например, каждое из деревьев, показанных ниже также является остовным деревом графа *G.*





Остовное дерево графа *G*

ТЕОРЕМА 1 Простой граф связен ⇔ он имеет остовное дерево .

Доказательство:

<= Сначала предположим, что простой граф *G* имеет остовное дерево *T* .

T содержит каждую вершину *G.*

В T существует путь между любыми двумя его вершинами.

Поскольку *T* является подграфом *G, в G* существует путь между любыми двумя его вершинами.

Следовательно, *G* связен.

Остовное дерево графа *G*

=> Теперь предположим, что *G* связен.

Если *G* не является деревом, он должен содержать простой цикл.

Удалим ребро в одном из этих простых циклов.

Полученный подграф имеет на одно ребро меньше, но по-прежнему содержит все вершины *G* и является связным.

Этот подграф по-прежнему связен, поскольку при удалении ребра из цикла, остается второй путь между вершинами, не содержащий это ребро.

Если этот подграф не является деревом, то он имеет простой цикл; поэтому, как и прежде, удаляем ребро, которое находится в простом цикле.

Остовное дерево графа *G*

Повторяем этот процесс до тех пор, пока не останется ни одного простого цикла.

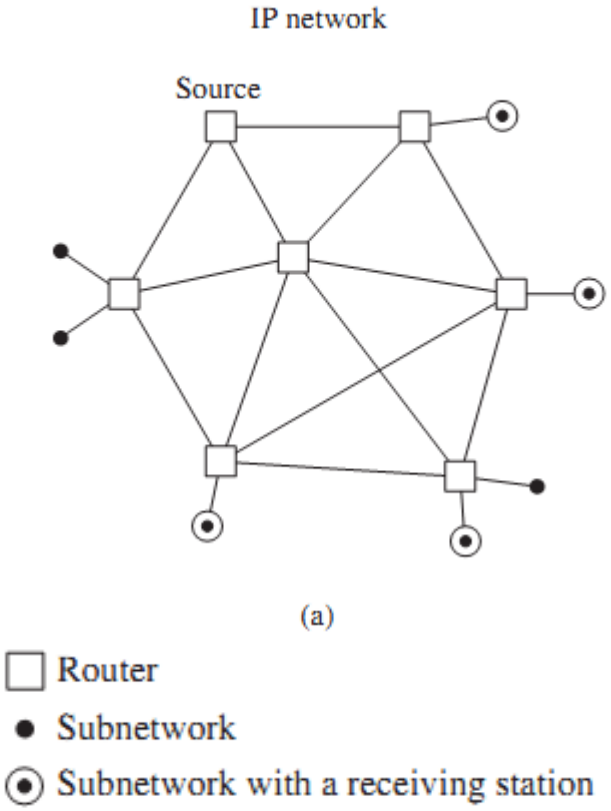
Это возможно, поскольку в графе имеется лишь конечное число ребер.

Процесс завершается, когда не остается простых циклов.

Дерево получается потому, что граф остается связным при удалении ребер.

Это дерево является остовным, поскольку из графа удалялись только ребра.

Остовное дерево графа *G*



ПРИМЕР Многоадресная IP-рассылка

Остовные деревья играют важную роль в многоадресной передаче по сетям Интернет-протокола (IP).

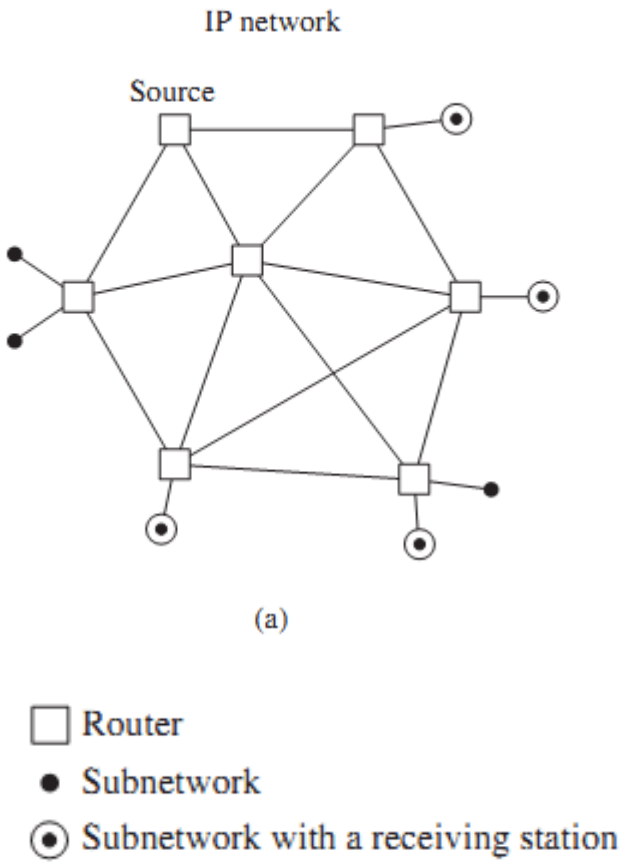
Чтобы отправить данные с исходного компьютера на несколько принимающих компьютеров, данные можно отправлять отдельно на каждый компьютер.

Такой тип сетевого взаимодействия называется одноадресной передачей.

Он неэффективен, поскольку по сети передается множество копий одних и тех же данных.

Чтобы сделать передачу данных на несколько принимающих компьютеров более эффективной, используется многоадресная IP-рассылка.

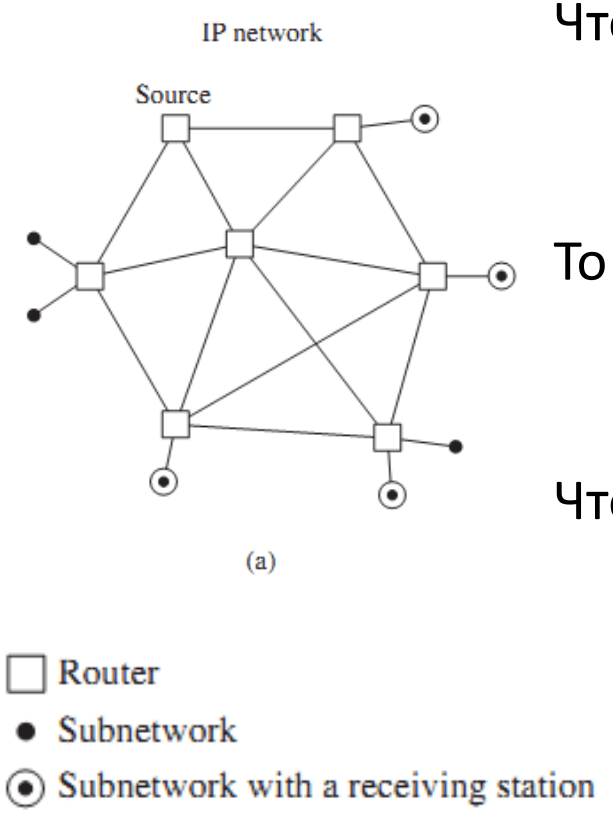
Остовное дерево графа *G*



При многоадресной IP-рассылке компьютер отправляет одну копию данных по сети, и когда данные достигают промежуточных маршрутизаторов, они пересылаются ≥ 1 другим маршрутизаторам, так что в конечном итоге все принимающие компьютеры в различных подсетях получают эти данные.

(Маршрутизаторы — это компьютеры, предназначенные для пересылки IP-датаграмм между подсетями в сети) .

Остовное дерево графа *G*

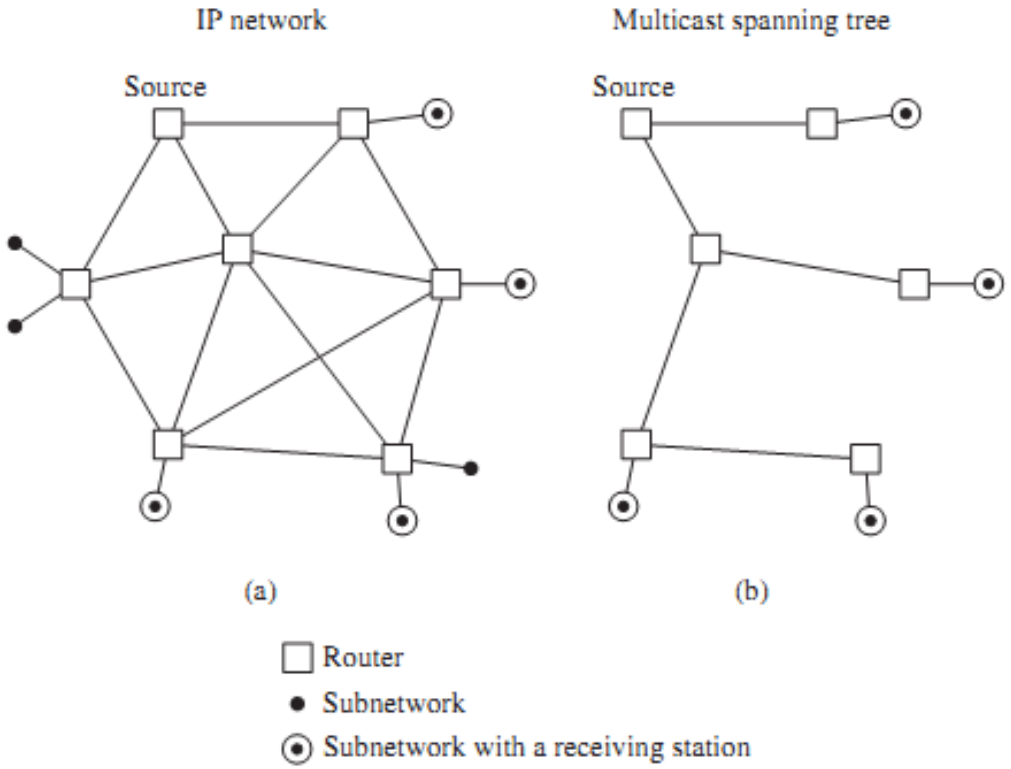


Чтобы данные достигали принимающих компьютеров как можно быстрее, на пути, по которому данные проходят по сети, не должно быть циклов.

То есть, как только данные достигли определенного маршрутизатора, они никогда не должны возвращаться на этот же маршрутизатор.

Чтобы избежать циклов, маршрутизаторы многоадресной рассылки используют сетевые алгоритмы для построения остовного дерева в графе, вершинами которого являются источник многоадресной рассылки , маршрутизаторы и подсети, содержащие принимающие компьютеры, а рёбра представляют собой связи между компьютерами и/или маршрутизаторами.

Остовное дерево графа *G*.



Корнем этого остовного дерева является источник многоадресной рассылки. Подсети, содержащие принимающие компьютеры, являются листьями дерева. (Обратите внимание, что подсети, не содержащие принимающих станций, в граф не включены.)

Остовное дерево графа *G.*

Доказательство теоремы 1 дает алгоритм для нахождения остовных деревьев путем удаления ребер из

простых циклов.

Этот алгоритм неэффективен, поскольку требует идентификации простых циклов.

Вместо того чтобы строить остовные деревья путем удаления ребер, остовные деревья можно строить путем последовательного добавления ребер .

Далее будут рассмотрены 2 алгоритма, основанные на этом принципе:

- Поиск в ширину
- Поиск в глубину

Остовное дерево графа *G*. Поиск в ширину (BFS)

Мы можем создать остовное дерево простого графа, используя поиск в ширину.

Будет построено корневое дерево, а основание этого корневого дерева образует остовное дерево.

Произвольно выбираем корень среди вершин графа.

Затем добавим все ребра, инцидентные этой вершине.

Новые вершины, добавленные на этом этапе, становятся вершинами уровня 1 в остовном дереве.

Произвольно упорядочим их.

Далее для каждой вершины на уровне 1, посещённой по порядку, добавляем каждое ребро, инцидентное этой вершине, в дерево, если оно не создаёт простой цикл.

Произвольно упорядочим дочерние элементы каждой вершины уровня 1.

Это создает вершины на уровне 2 дерева.

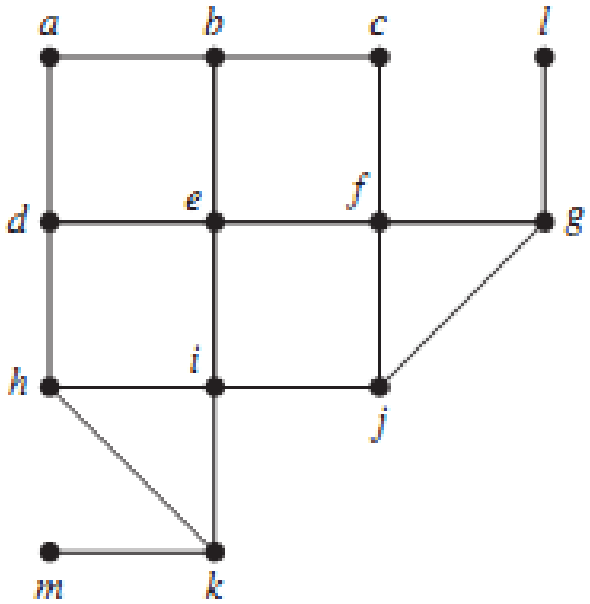
Повторяем ту же процедуру, пока не будут добавлены все вершины дерева.

Процедура заканчивается, поскольку в графе имеется только конечное число ребер .

Остовное дерево получается, поскольку мы создали дерево, содержащее каждую вершину графа.

Остовное дерево графа *G*. Поиск в ширину (BFS)

ПРИМЕР. Использовать поиск в ширину, чтобы найти остовное дерево для графа, показанного ниже.



Граф *G*.

Остовное дерево графа *G*. Поиск в ширину (BFS)

Решение: В качестве корня выбираем вершину *e* .

Затем добавляем ребра, инцидентные всем вершинам, смежным с *e*, так что добавляются ребра из *e* в *b, d, f* и *i* .

Эти 4 вершины находятся на уровне 1 дерева.

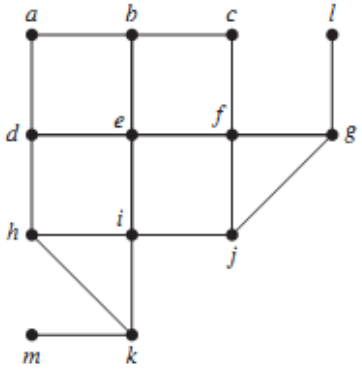
Затем добавим ребра, инцидентные вершинам уровня 1, к соседним вершинам, которых еще нет в дереве.

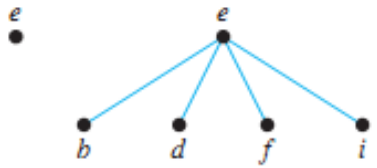
Добавляются ребра из *b* в *a* и *c, а также ребра из d в h*, из *f в j* и *g* и из *i в k* .

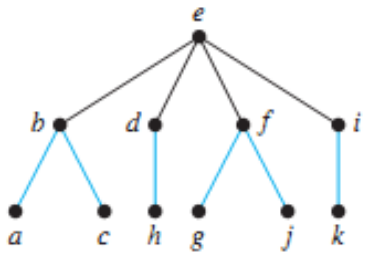
Новые вершины *a, c, h, j, g* и *k* находятся на уровне 2.

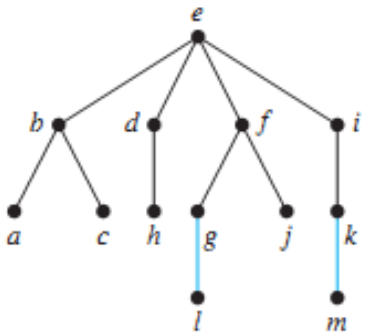
Затем добавим ребра из этих вершин в соседние вершины, которых еще нет в графе.

Добавляются ребра из *g в l* и из *k в m* .









Остовное дерево графа *G.* Поиск в ширину.

ALGORITHM 1 Breadth-First Search.
procedure BFS1 (*G*: connected graph with vertices v_1, v_2, ..., v_n)
T := tree consisting only of vertex v_1
L := empty list
put v_1 in the list *L* of unprocessed vertices
while *L* is not empty
    remove the first vertex, *v*, from *L*
    for each neighbor *w* of *v*
        if *w* is not in *L* and not in *T* then
            add *w* to the end of the list *L*
            add *w* and edge {*v, w*} to *T*

Расширенный поиск в ширину (BFS2)

Процедура поиска в ширину BFS2, представленная ниже, предполагает, что входной граф $G = (V, E)$ представлен с использованием списков смежности.

Он связывает несколько дополнительных атрибутов с каждой вершиной графа G.

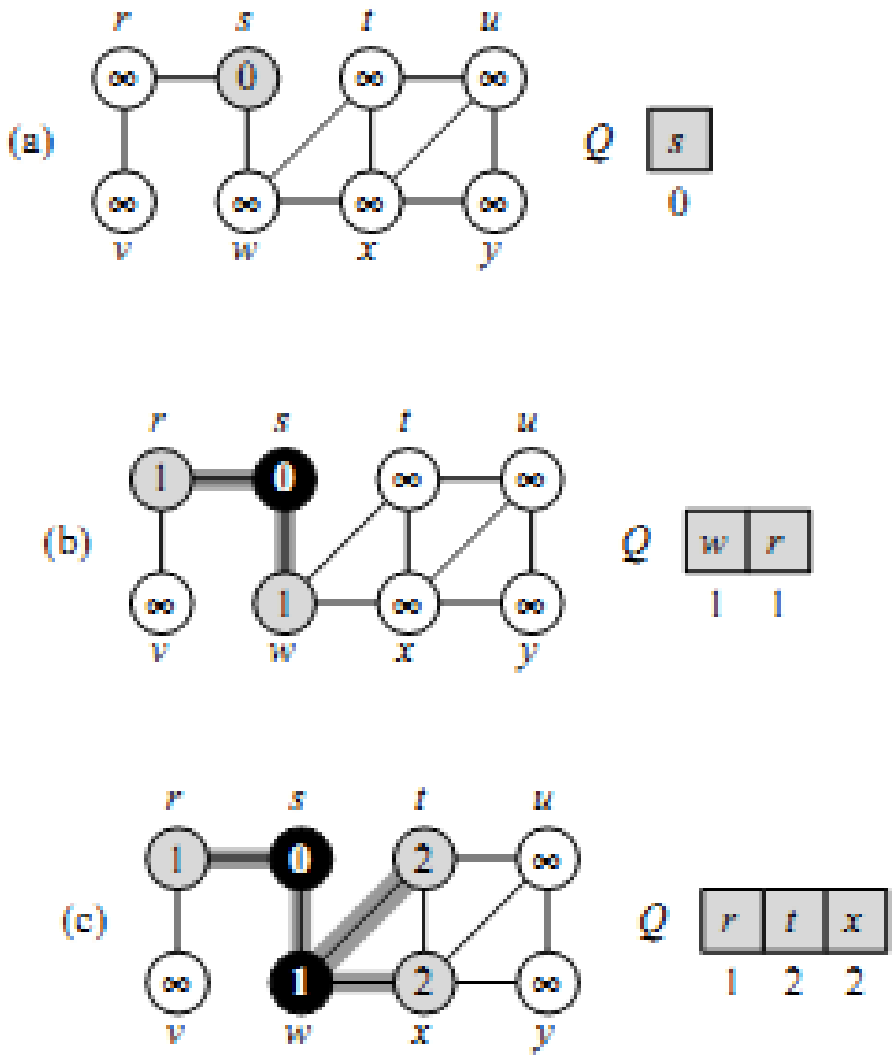
Будем хранить цвет каждой вершины $u \in V$ в атрибуте $u.color$ и предшественника u в атрибуте $u.\pi$

Если u не имеет предшественника (например, если $u = s$ или u не была обнаружена), то $u.\pi$ = NIL.

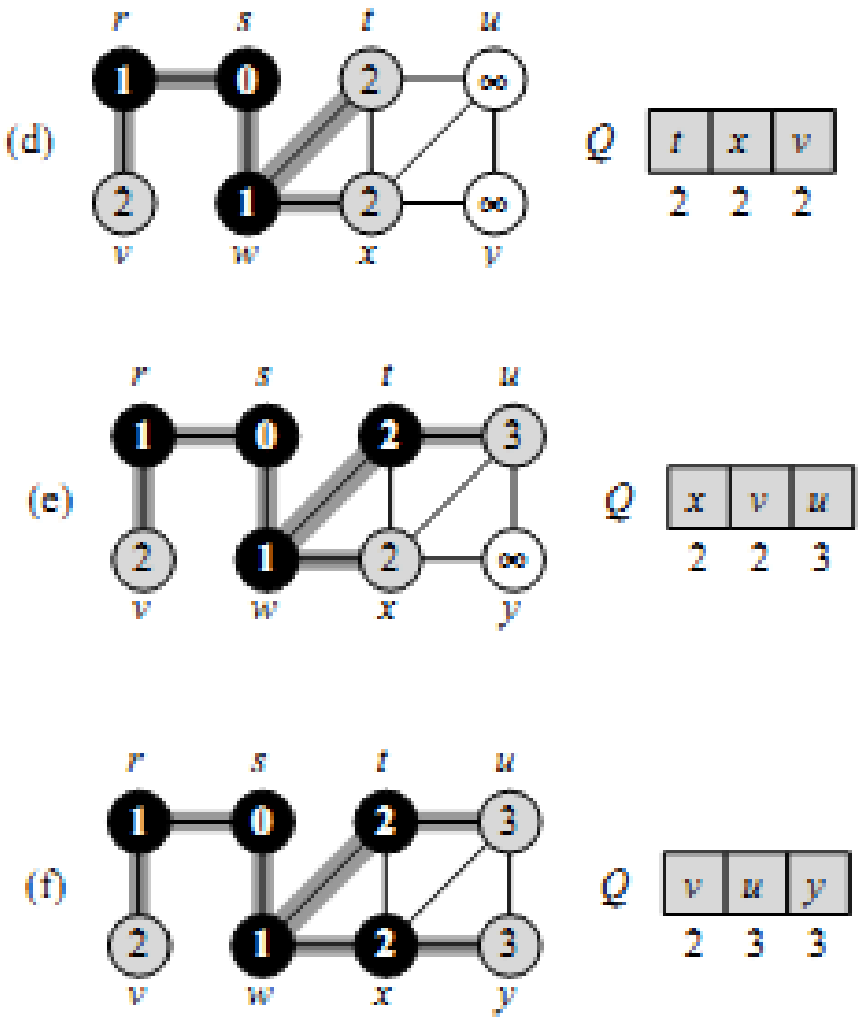
Атрибут $u.d$ будет содержать расстояние от источника s до вершины u, вычисленное алгоритмом.

Алгоритм также использует *FIFO*-очередь Q для управления множеством серых вершин.

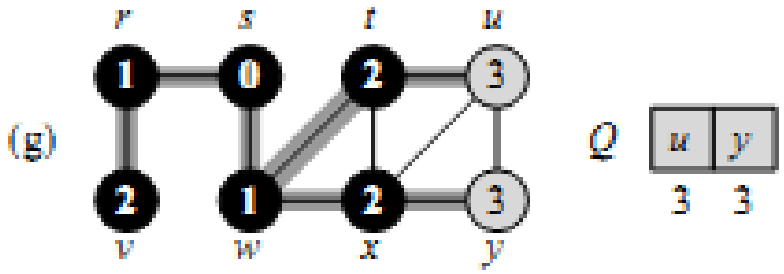
```
BFS2(G, s)
1 for each vertex u ∈ G.V - {s}
2       u.color = WHITE
3 .     u.d = ∞
4       u.π = NIL
5 s.color = GRAY
6 s.d = 0
7 s.π = NIL
8 Q =∅ ;
9 ENQUEUE(Q, s)
10 while Q ≠∅ ;
11        u = DEQUEUE(Q)
12        for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                v.color = GRAY
15                v.d = u.d + 1
16                v.π = u
17                ENQUEUE(Q, v)
18        u.color = BLACK
```

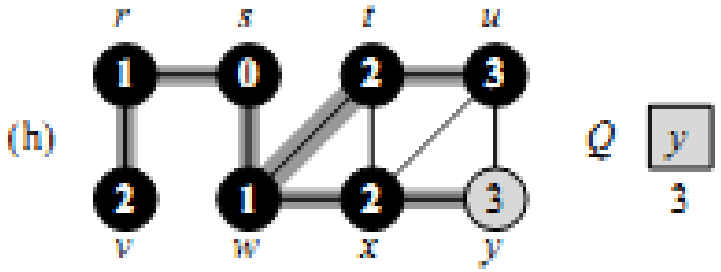


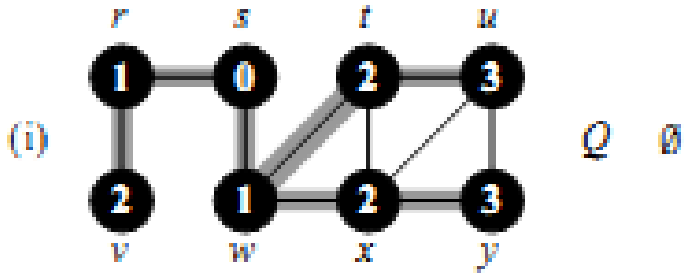

```
BFS2(G, s)
1 for each vertex u ∈ G.V - {s}
2       u.color = WHITE
3 .     u.d = ∞
4       u.π = NIL
5 s.color = GRAY
6 s.d = 0
7 s.π = NIL
8 Q =∅ ;
9 ENQUEUE(Q, s)
10 while Q ≠∅ ;
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```




```
BFS2(G, s)
1 for each vertex u ∈ G.V - {s}
2       u.color = WHITE
3 .     u.d = ∞
4       u.π = NIL
5 s.color = GRAY
6 s.d = 0
7 s.π = NIL
8 Q =∅ ;
9 ENQUEUE(Q, s)
10 while Q ≠∅ ;
11        u = DEQUEUE(Q)
12        for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                v.color = GRAY
15                v.d = u.d + 1
16                v.π = u
17                ENQUEUE(Q, v)
18        u.color = BLACK
```








```
BFS2(G, s)
1 for each vertex u ∈ G.V - {s}
2       u.color = WHITE
3 .     u.d = ∞
4       u.π = NIL
5 s.color = GRAY
6 s.d = 0
7 s.π = NIL
8 Q =∅ ;
9 ENQUEUE(Q, s)
10 while Q ≠∅ ;
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Инвариант :
В тесте в строке 10 очередь *Q* состоит из множества серых вершин.

Инвариант цикла сохраняется, поскольку всякий раз, когда вершина окрашивается в серый цвет (в строке 14), она также ставится в очередь (в строке 17), а всякий раз, когда вершина извлекается из очереди (в строке 11), она также окрашивается в черный цвет (в строке 18).

Результаты поиска в ширину могут зависеть от порядка, в котором посещаются соседи данной вершины в строке 12:

Дерево поиска в ширину может варьироваться, но расстояния *d,* вычисленные алгоритмом, не меняются.

Анализ

Прежде чем доказывать различные свойства поиска в ширину, давайте проанализируем время его выполнения на входном графе $G = (V, E)$.

После инициализации BFS вершина никогда не становится белой, и, значит, тест в строке 13 гарантирует, что каждая вершина попадает в очередь один раз, и, следовательно, выводится из очереди один раз.

Операции постановки в очередь и извлечения из очереди занимают $O(1)$ времени, поэтому общее время, затрачиваемое на операции с очередью, составляет $O(V)$.

Поскольку процедура сканирует список смежности каждой вершины только тогда, когда вершина извлекается из очереди, она сканирует каждый список смежности один раз.

Поскольку сумма длин всех списков смежности равна $\Theta(E)$, общее время, затрачиваемое на сканирование списков смежности, равно $O(E)$.

Накладные расходы на инициализацию составляют $O(V)$, и, значит, общее время выполнения процедуры BFS составляет $O(V+E)$.

Таким образом, поиск в ширину выполняется за время, линейно зависящее от размера представления списка смежности графа

Кратчайшие пути в графе

BFS2 находит расстояние до каждой достижимой вершины в графе $G = (V, E)$ от заданной исходной вершины $s \in B$.

Определим кратчайшее расстояние $\delta(s, v)$ от s до v как минимальное количество ребер в любом пути от вершины s до вершины v;

если пути из s в v нет, то $\delta(s, v) = \infty$.

Назовем путь длины $\delta(s, v)$ из s в v кратчайшим путем из s в v .

Кратчайшие пути в графе

Лемма 2 Пусть $G = (V, E)$ — ориентированный или неориентированный граф, и пусть $s \in V$ — произвольная вершина.

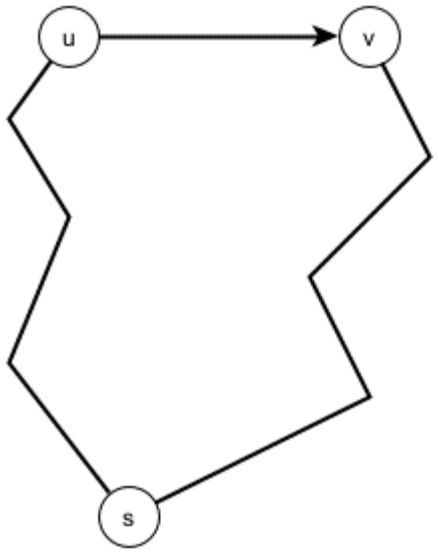
Тогда для любого ребра $(u, v) \in E$,

$\delta(s, v) \leq \delta(s, u) + 1$.

Доказательство. Если u достижима из s , то v тоже достижима.

В этом случае кратчайший путь из s в v не может быть длиннее кратчайшего пути из s в u , за которым следует ребро (u, v), и, значит, неравенство выполняется.

Если u недостижима из s, то $\delta(s, u) = \infty$, и неравенство выполняется.



Кратчайшие пути в графе

Мы хотим показать, что BFS2 правильно вычисляет $v.d = \delta(s, v)$ для каждой вершины $v \in V$.

Покажем, что $v.d$ ограничивает $\delta(s, v)$ сверху.

Лемма 3 Пусть $G = (V, E)$ BFS— ориентированный или неориентированный граф, и предположим, что выполняется на G из заданной исходной вершины $s \in V$.

Тогда по завершении BFS для каждой вершины $v \in V$, значение $v.d$, вычисленное BFS, удовлетворяет неравенству

$v.d \geq \delta(s, v)$.

Доказательство Мы используем индукцию по количеству операций ENQUEUE.

Индуктивная гипотеза заключается в том, что $v.d \geq \delta(s, v)$ для всех $v \in V$.

1) Основой индукции является ситуация сразу после постановки s в очередь в строке 9 BFS2.

Индуктивная гипотеза здесь верна, потому что $s.d = 0 = \delta(s, s)$ и

$v.d = \infty \geq \delta(s, v)$ $\forall$ $v \in \{V - s\}$.

Кратчайшие пути в графе

В качестве шага индукции рассмотрим белую вершину v , обнаруженную во время поиска из вершины u .

Индуктивная гипотеза подразумевает, что $u.d \geq \delta(s, u)$.

Из присваивания, выполненного в строке 15 BFS2 и из Леммы 2, получаем

$v.d = u.d + 1$

$\geq \delta(s, u) + 1$

$\geq \delta(s, v)$.

Вершина v попадает в очередь, и она больше никогда не попадет в очередь снова, поскольку она также становится серой, а предложение **then** в строках 14–17 выполняется только для белых вершин.

Таким образом, значение $v.d$ больше никогда не изменится, и индуктивная гипотеза сохраняется.

Кратчайшие пути в графе

Чтобы доказать, что $v.d = \delta(s, v)$, мы должны более подробно рассмотреть, как функционирует очередь Q в ходе BFS2 (для коллоквиума).

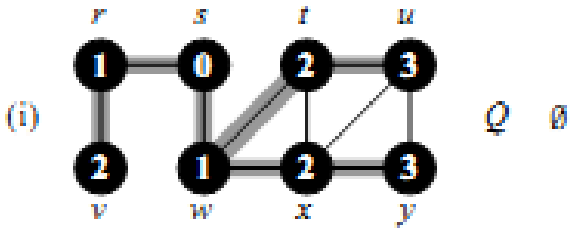
Теорема 4 (Корректность BFS-2)

Пусть $G = (V, E)$ — ориентированный или неориентированный граф, и предположим, что BFS2 запускается на G из заданной исходной вершины $s \in V$.

Тогда во время выполнения, BFS2 обнаруживает каждую вершину $v \in V$, достижимую из источника s, и по завершении BFS2

$v.d = \delta(s, v) \; \forall v \in V$.

Деревья поиска в ширину ,



Процедура BFS2 строит дерево поиска в ширину при поиске по графу.

Ребра дерева соответствует атрибутам π .

Более формально, для графа $G = (V, E)$ с источником s мы определяем подграф предшествования как $G_\pi = (V_\pi, E_\pi)$,

где $V_\pi = \{ v \in V : v.\pi \neq NIL \} \cup \{s\}$

и $E_\pi = \{ (v.\pi, v) : v \in V_\pi - \{ s \} \}$.

Подграф предшествования G_π является деревом поиска в ширину, если V состоит из вершин, достижимых из s, и для всех $v \in V$, подграф G_π содержит единственный простой путь из s в v, который также является кратчайшим путем из s в v в G.

Дерево поиска в ширину на самом деле является деревом, поскольку оно связно и $|E| = |V| - 1$.

Мы называем ребра в E древесными ребрами.

Следующая лемма показывает, что подграф предшествования, созданный BFS2 представляет собой дерево поиска в ширину .

Деревья поиска в ширину

Лемма 5. При применении к ориентированному или неориентированному графу $G = (V, E)$ BFS2 строит атрибут π так, что подграф предшествования $G_\pi = (V_\pi, E_\pi)$ является деревом поиска в ширину.

Доказательство. Строка 16 алгоритма BFS2 устанавливает

что $v.\pi = u \Leftrightarrow (u, v) \in E$, и

$\delta(s, v) < \infty$ — то есть, если v достижима из s

— таким образом, V_π состоит из вершин в V, достижимых из s.

Поскольку G_π образует дерево, оно содержит уникальный простой путь из s в каждую вершину в V_π.

Применяя теорему 5 индуктивно, заключаем, что каждый такой путь является кратчайшим путем в графе G.

Остовное дерево графа *G*. Поиск в глубину

Мы можем построить остовное дерево для связного простого графа, используя поиск в глубину (Depth First Search, DFS) .

DFS сформирует корневое дерево, а остовное дерево будет основанием этого корневого дерева.

Произвольно выберем вершину графа в качестве корня.

Сформируем путь, начинающийся из выбранной вершины путем последовательного добавления вершин и ребер, где каждое новое ребро инцидентно предыдущей вершине пути и вершине, которой еще нет в пути.

Продолжим добавлять вершины и ребра к этому пути как можно дольше.

Если путь проходит через все вершины графа, то дерево, состоящее из этого пути представляет собой остовное дерево.

Однако, если путь не проходит через все вершины, надо добавить больше вершин и ребер.

Остовное дерево графа *G*. Поиск в глубину

Надо вернуться к предпоследней вершине пути и, если возможно, сформировать новый путь, начинающийся в этой вершине и проходящий через вершины, которые еще не были посещены.

Если это невозможно, переместиться назад еще одну вершину пути, то есть на 2 вершины пути, и попробовать еще раз.

Повторим эту процедуру, начиная с последней посещенной вершины, двигаясь назад по пути на один шаг по одной вершине за раз, формируя новые пути, которые являются настолько длинными, насколько это возможно, пока больше ребер не может быть добавлено.

Поскольку граф имеет конечное число ребер и является связным, этот процесс заканчивается созданием остовного дерева.

Каждая вершина, которая является конечной на некотором этапе алгоритма, будет быть листом в корневом дереве, и

каждая вершина, где начинает строиться новый путь, будет внутренней вершиной.

Остовное дерево графа *G*. Поиск в глубину

а) Обратите внимание на рекурсивный характер этой процедуры.

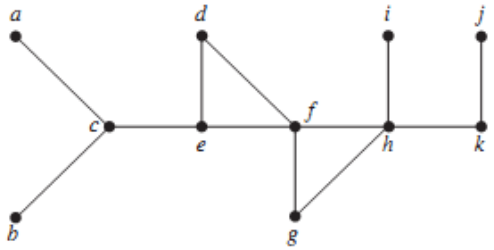
б) Также обратите внимание, что если вершины в графе упорядочены, то выбор ребер на каждом этапе процедуры определяется однозначно, мы всегда выбираем первую вершину в заданном порядке.

Однако мы не всегда будем явно упорядочивать вершины графа.

Поиск в глубину также называется бэктрэкингом (откатом), поскольку алгоритм возвращается к ранее посещенным вершинам для добавления путей.

Приведенный ниже пример иллюстрирует бэктрэкинг.

ПРИМЕР Используйте поиск в глубину, чтобы найти остовное дерево



Остовное дерево графа *G*. Поиск в глубину

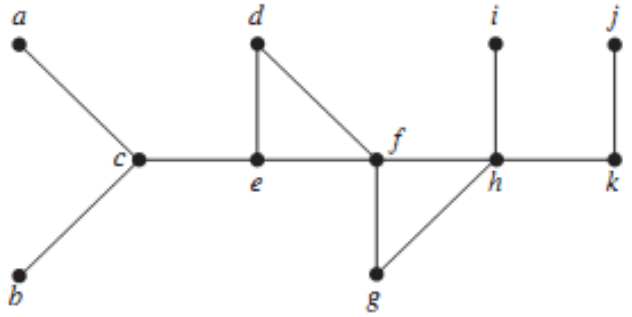
Решение: Шаги, используемые при поиске в глубину для создания остовного дерева *G*, показаны ниже.

Мы произвольно начинаем с вершины *f* .

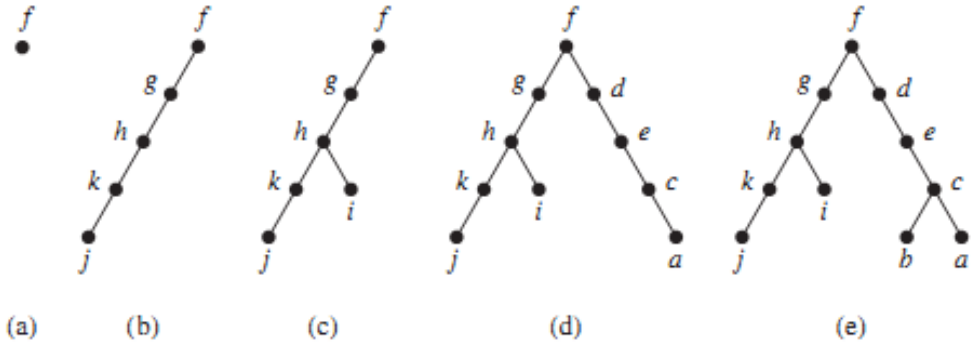
Путь строится путем последовательного добавления ребер, инцидентных вершинам, еще не входящим в путь, пока это возможно.

Сначала создается путь *f, g, h, k, j* (обратите внимание, что можно было бы построить и другие пути).

Далее откатываемся к *k* .



Граф *G*.



Поиск в глубину в графе *G*.

Остовное дерево графа *G*. Поиск в глубину

Не существует пути, начинающегося в *k* и содержащего вершины, которые еще не посещены.

Значит, откатываемся к *h* .

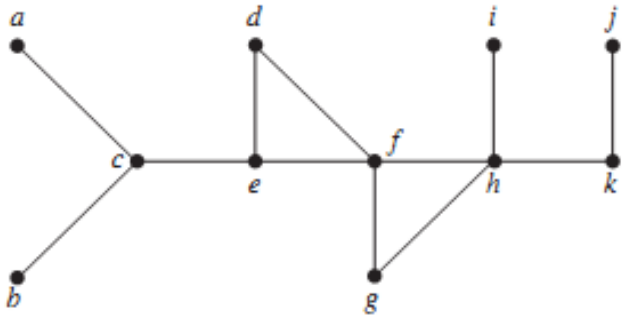
Образуем путь *h, i* .

Затем откатываемся в *h*, а затем в *f* .

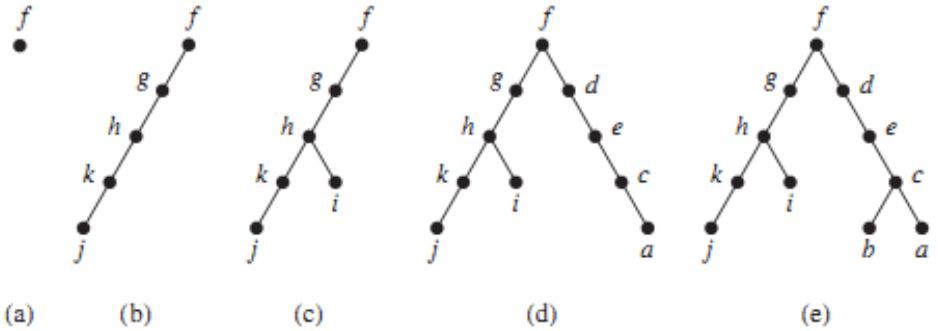
Из *f* строим путь *f, d, e, c, a* .

Затем откатываемся в *c* и образуем путь *c, b* .

В результате получается остовное дерево.



Граф *G*.



Поиск в глубину в графе *G*.

Остовное дерево графа *G.* Поиск в глубину

Мы объяснили, как найти остовное дерево графа с помощью поиска в глубину.

Чтобы прояснить рекурсивную природу алгоритма, нам понадобится немного терминологии.

Мы говорим, что мы осуществляем поиск из вершины *v,* когда мы выполняем шаги поиска в глубину, начинающегося с момента когда *v* добавляется в дерево и завершающегося, когда мы вернулись к *v* в последний раз .

Ключевое наблюдение, необходимое для понимания рекурсивной природы алгоритма, заключается в том, что когда мы добавляем ребро, соединяющее вершину *v* с вершиной *w*, мы начинаем новый поиск из вершины *w.*

И мы заканчиваем поиск из *w,* прежде чем вернуться в *v* для завершения поиска из *v* .

Остовное дерево графа *G*. Поиск в глубину

В алгоритме DFS мы строим остовное дерево графа *G* с вершинами v_1, ..., v_n сначала выбрав вершину v_1 в качестве корня.

Первоначально мы задали *T* как дерево, содержащее только v_1.

На каждом шаге мы добавляем новую вершину к дереву *T* вместе с ребром от вершины, уже имеющейся в *T,* к этой новой вершине и ведем поиск из этой новой вершины.

Обратите внимание, что по завершении алгоритма *T* не содержит простых циклов, поскольку не добавляется ни одного ребра, соединяющего 2 вершины, уже находящиеся в дереве.

Более того, *T* остается связным по мере его создания.

(Последние два наблюдения можно легко доказать с помощью математической индукции.)

Поскольку *G* связен, каждая вершина в *G* посещается алгоритмом и добавляется в дерево.

Из этого следует, что *T* является остовным деревом графа *G.*

Остовное дерево графа *G*. Поиск в глубину

ALGORITHM 3 DFS1.

procedure DFS1(*G*: connected graph with vertices $v_1, v_2, \ldots, v_n$)

T := tree consisting only of the vertex v_1

visit(v_1)

procedure *visit*(*v*: vertex of *G*)

for each vertex *w* adjacent to *v* and not yet in *T*

add vertex *w* and edge {*v*, *w*} to *T*

visit(*w*)

Остовное дерево графа *G.* Поиск в глубину

Поиск в глубину может быть использован в качестве основы для алгоритмов, решающих множество различных задач.

Например, его можно использовать:

- Для поиска путей и циклов в графе,
- Для выделения связных компонент графа,
- Для поиска точек сочленения связного графа.

Как мы увидим, поиск в глубину является основой методов бэктрэкинга, используемых для поиска решения вычислительно сложных задач.

Расширенный поиск в глубину (DFS2)

Как и в BFS2, всякий раз, когда DFS2 обнаруживает вершину *v* во время просмотра списка смежности уже обнаруженной вершины *u,* он регистрирует это событие, устанавливая атрибут предшественника вершины *v*, $v.\pi = u$.

В отличие от BFS2, подграф предшествования которого образует дерево, подграф предшествования, созданный DFS2, может состоять из нескольких деревьев , поскольку поиск может повторяться из нескольких источников.

Поэтому мы определяем подграф предшествования DFS2 немного иначе, чем в BFS:

$G_\pi = (V, E_\pi)$,

где $E_\pi = \{(v.\pi, v) : v \in V \text{ и } v.\pi \neq \text{NIL}\}$.

Подграф предшествования DFS2 образует лес поиска в глубину, состоящий из нескольких деревьев поиска в глубину .

Ребра в E_π являются древесными ребрами.

Поиск в глубину DFS2

Как и в BFS2, DFS2 раскрашивает вершины во время поиска, чтобы обозначить их состояние.

Каждая вершина первоначально белая, в момент обнаружения в процессе поиска вершина становится серой, а затем она становится черной, когда поиск из этой вершины завершен, то есть когда ее список смежности полностью просмотрен.

Этот метод гарантирует, что каждая вершина попадает ровно в 1 дерево поиска в глубину, так что эти деревья не пересекаются.

Каждая вершина *v* имеет 2 метки времени :

первая метка *v.d* записывает, когда *v* обнаружена в первый раз (и становится серой),

вторая метка *v.f* записывает, когда поиск в глубину завершает проверку списка смежности вершины *v* (и делает вершину *v* черной) .

Эти временные метки предоставляют важную информацию о структуре графа.

Поиск в глубину DFS2

Временные метки представляют собой целые числа от 1 до 2|*V*|, *поскольку* для каждой из вершин |*V*| существует 1 событие обнаружения и 1 событие завершения обработки.

Для каждой вершины *u*,

$u.d < u.f$ (1)

Вершина *u является:*

БЕЛОЙ перед моментом времени $u.d$,

СЕРОЙ между временем $u.d$ и временем $u.f$, и

ЧЁРНОЙ после этого .

Следующий псевдокод представляет собой базовый алгоритм DFS2.

Входной граф *G* может быть неориентированным или ориентированным.

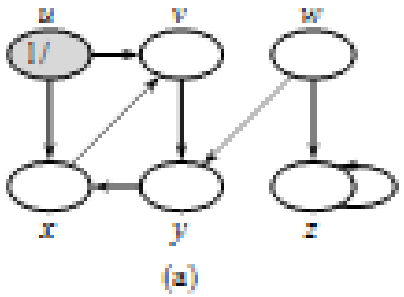
Переменная *time* — это глобальная переменная, которую мы используем для отметки времени.

Поиск в глубину. Алгоритм 4 (DFS2)

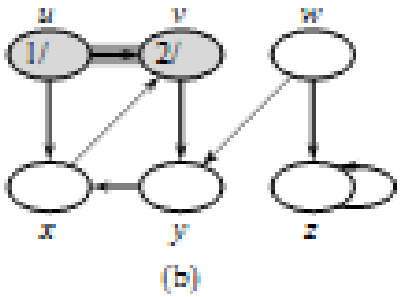
```
DFS2(G)
1 for each vertex u ∈ G.V
2       u.color = WHITE
3       u.π =  NIL
4 time = 0
5 for each vertex u ∈ G.V
6      if u.color == WHITE
7           DFS-VISIT(G, u)

DFS-VISIT (G, u)
1 time =time + 1              // белая вершина u только что обнаружена
2 u.d = time
3 u.color = GRAY
4 for each  v ∈ G.Adj[u]      // просмотр ребра (u, v) /
5       if v.color == WHITE
6             v.π = u
7       DFS-VISIT(G, v)
8 u.color =BLACK              // вершина u становится черной; ее обработка завершена
9 time = time + 1
10 u. f = time
```

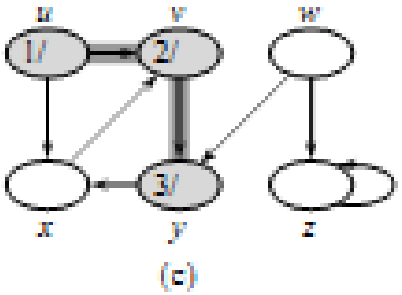
ПРИМЕР:DFS2



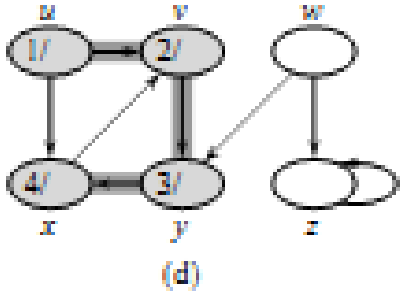
time = 1



time = 2

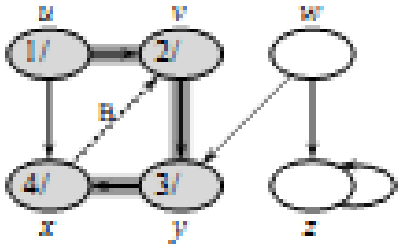


time = 3



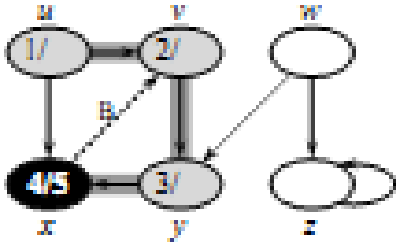
(d)

time = 4



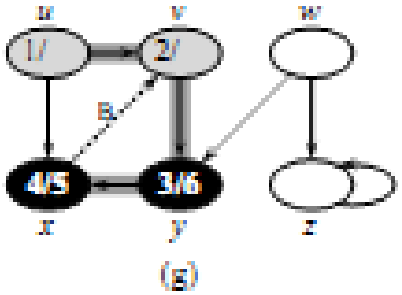
(e)

time = 4



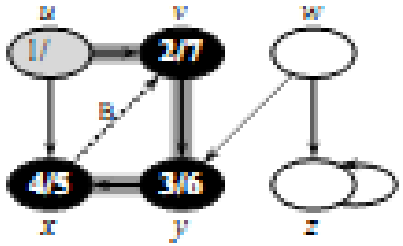
(f)

time = 5



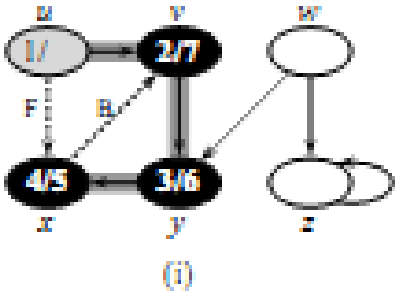
(g)

time = 6



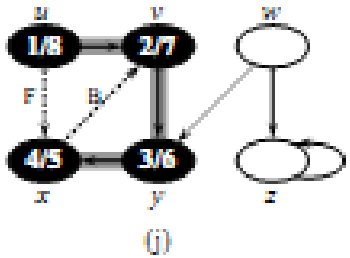
(h)

time = 7



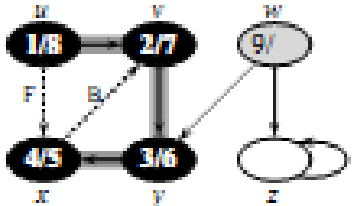
(i)

time = 7



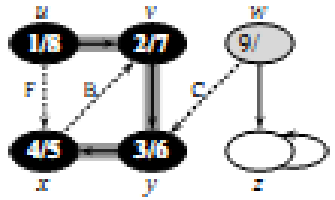
(j)

time = 8



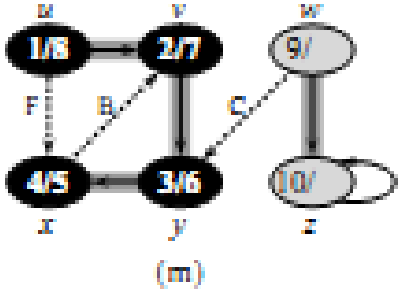
(k)

time = 9



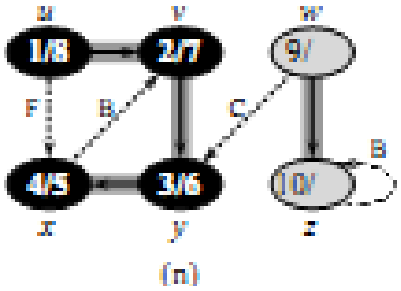
(l)

time = 9



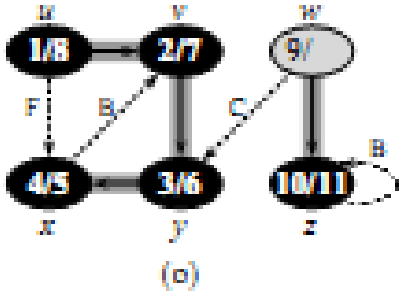
(m)

time = 10



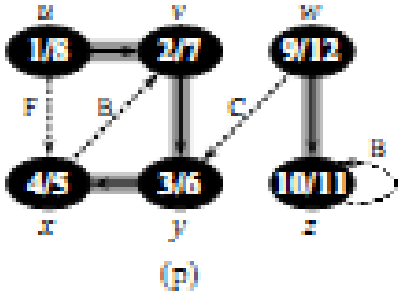
(n)

time = 10



(o)

time = 11



(p)

time = 12

Обратите внимание, что результаты DFS2 могут зависеть от порядка, в котором строка 5 DFS2 проверяет вершины, и от порядка, в котором строка 4 DFS-VISIT посещает соседей вершины.

Свойства DFS2

DFS2 дает ценную информацию о структуре графа.

Возможно, самым основным свойством DFS является то, что подграф предшествования G_π действительно образует лес деревьев , поскольку структура деревьев поиска в глубину в точности отражает структуру рекурсивных вызовов DFS-VISIT.

1. То есть, $u = v.\pi$ (вершина *u* является отцом вершины *v)* : ⇔

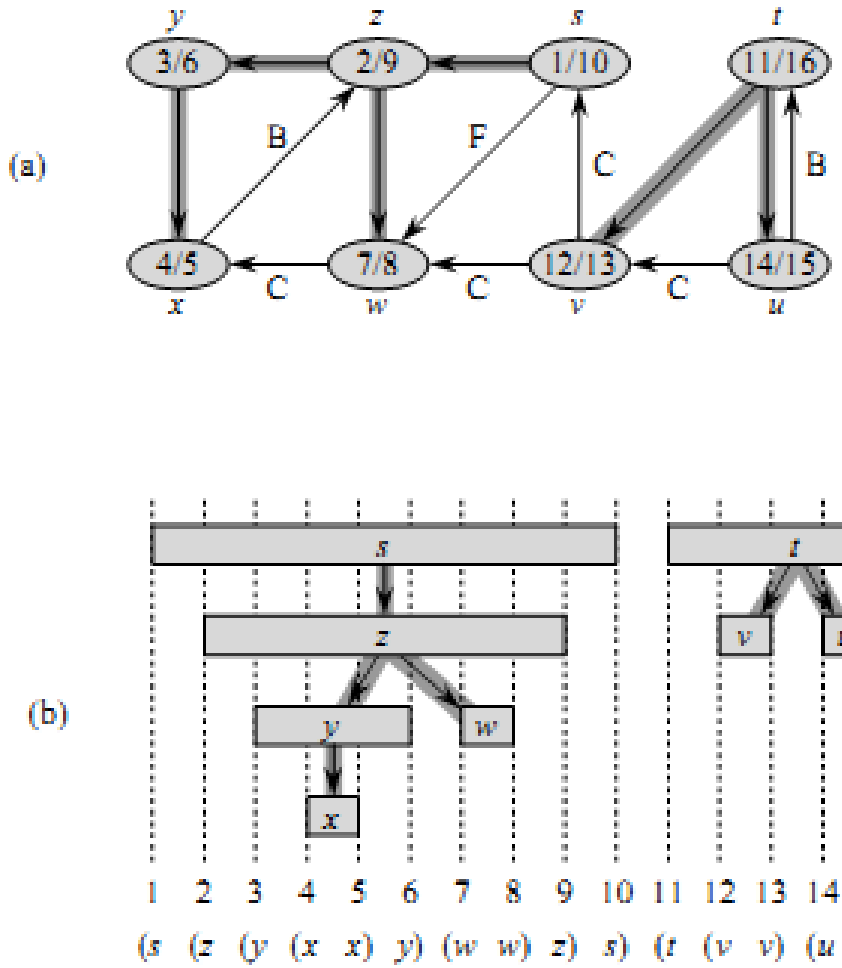
DFS-VISIT(*G, v*) был вызван во время поиска в списке смежности *u* .

2. Кроме того, вершина *v* является потомком вершины *u* в глубинном остовном лесу ⇔ *v* обнаружена в то время, когда *u* является серой.

3. Еще одним важным свойством DFS2 является то, что время обнаружения и завершения имеет скобочную структуру .

Если мы представим открытие вершины *u* с помощью левой скобки «(*u* », а ее завершение — с помощью правой скобки « *u*) », то история открытий и завершений образует правильно построенное выражение в том смысле, что скобки правильно вложены.

Например, DFS2 на рисунке (а) соответствует скобкам, показанным на рисунке (б).



(b) Интервалы времени открытия и времени завершения каждой вершины соответствуют показанным скобкам.
Каждый прямоугольник охватывает интервал, заданный временем открытия и завершения соответствующей вершины.
Показаны только ребра деревьев .
Если 2 интервала перекрываются, то один из них вложен в другой,
а вершина, соответствующая меньшему интервалу, является потомком вершины, соответствующей большему.

Следующая теорема дает еще один способ охарактеризовать структуру скобок.

Теорема 6 (Теорема о скобках)

В любом DFS2 графе (ориентированном или неориентированном)

$G = (V, E)$ для любых 2 вершин u и v выполняется ровно 1 из следующих 3 условий:

интервалы $[u.d, u.f]$ и $[v.d, v.f]$ полностью не пересекаются, и ни u , ни v *не* являются потомками друг друга в глубинном остовном лесу,

интервал $[u.d, u.f]$ полностью содержится в интервале $[v.d, v.f]$, а u является потомком v в дереве поиска в глубину,

или

интервал $[v.d, v.f]$ полностью содержится в интервале $[u.d, u.f]$, а v является потомком u в дереве поиска в глубину.

Доказательство Начнем со случая, когда $u.d < v.d$.

Мы рассматриваем 2 подслучая,

в зависимости от того, $v.d < u.f$ или нет.

1) $v.d < u.f$,

Вершина *v* была обнаружена, когда *u* была еще серой ,

что подразумевает, что *v* является потомком *u* .

Более того, поскольку *v* была обнаружена позже, чем *u*, все исходящие ребра *v* исследуются, и *v* завершается, прежде чем поиск возвращается *в u* и завершает обработку *u*.

В этом случае, следовательно, интервал [*v.d , v.f*] целиком содержится внутри интервала [*u.d, u.f*].

2) В случае, когда $u.f < v.d$, и по неравенству (1),

$u.d < u.f < v.d < v.f$;

таким образом, интервалы [*u.d, u.f*] и [*v.d, v.f*] не пересекаются.

Поскольку интервалы не пересекаются,

ни одна вершина не была обнаружена, то время как другая была серой,

и поэтому ни одна из вершин не является потомком другой.

Случай, когда $v.d < u.d$, аналогичен, но роли *u* и *v* в приведенном выше рассуждении меняются местами.

Свойства DFS2

Следствие 7

Вершина *v* является собственным потомком вершины *u* в глубинном остовном лесу для (ориентированного или неориентированного) графа $G \Leftrightarrow u.d < v.d < v.f < u.f$.

Доказательство следует непосредственно из теоремы 6.

Свойства DFS2 Классификация ребер

Еще одним интересным свойством DFS2 является то, что поиск можно использовать для классификации ребер входного графа $G = (V, E)$.

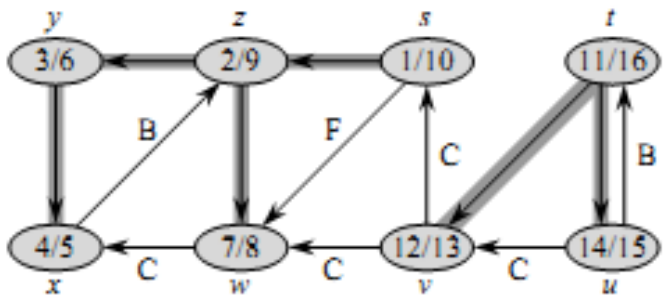
Тип каждого ребра может предоставить важную информацию о графе.

Например, ориентированный граф ацикличен ⇔ DFS не выдает «обратных» ребер.

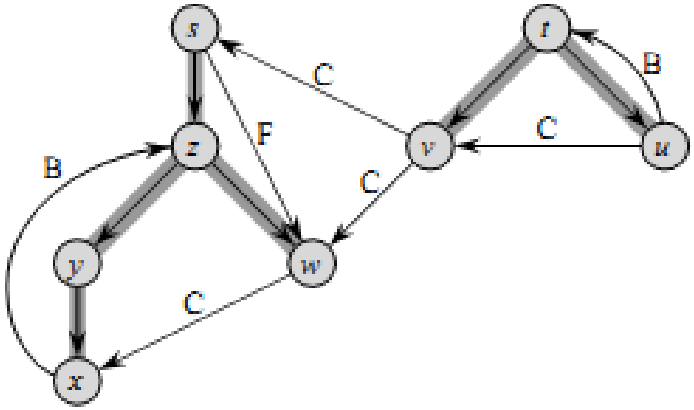
Мы можем определить 4 типа ребер в терминах леса глубины G_π произведенный DFS2 на ориентированном графе G :

1. Древесные ребра — это ребра в глубинном остовном лесу G_π. Ребро (u, v) является древесным ребром, если v была впервые обнаружена при исследовании ребра (u, v).
2. Обратные ребра — это те ребра (u, v), которые соединяют вершину u с предком в дереве поиска в глубину. Мы считаем петли, которые могут встречаться в ориентированных графах, обратными ребрами.
3. Прямые ребра — это те недревесные ребра u, v), которые соединяют вершину u с потомком в дереве поиска в глубину.
4. Поперечные ребра — все остальные ребра. Они могут проходить между вершинами в одном и том же дереве поиска в глубину, пока одна вершина не является предком другой, или они могут проходить между вершинами в разных деревьях поиска в глубину.

Свойства DFS2 Классификация ребер



(а)



(б)

- На рисунке (б) также показано, как перерисовать граф рисунка (а) так, чтобы все древесные и прямые ребра были направлены вниз в дереве поиска в глубину, а все обратные ребра были направлены вверх.

Свойства DFS2 Классификация ребер

Алгоритм DFS2 имеет достаточно информации для классификации некоторых ребер по мере их обнаружения.

Основная идея заключается в том, что когда мы впервые исследуем ребро (*u, v*), цвет вершины *v* говорит нам кое-что о ребре:

1. БЕЛЫЙ цвет обозначает древесное ребро ,
2. СЕРЫЙ указывает наобратное ребро, а
3. ЧЕРНЫЙ цвет обозначает прямое или поперечное ребро.

Свойства DFS2 Классификация ребер

Теперь мы покажем, что прямые и поперечные ребра никогда не встречаются в поиске в глубину на неориентированном графе .

Теорема 8

При поиске в глубину на неориентированном графе *G* каждое ребро *G* является либо древесным ребром, либо обратным ребром.

Доказательство. Пусть {*u, v*} — произвольное ребро графа *G*, и предположим без ограничения общности, что $u.d < v.d$.

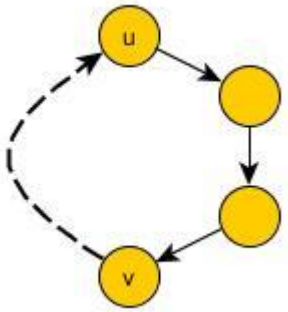
Тогда поиск должен обнаружить и завершить *v* до того, как он завершит обработку *u* (пока *u* серая), поскольку *v* находится в списке смежности *u* .

1) Если в первый раз поиск исследует ребро { *u, v*} в направлении от *u* к *v* , то *v* остается необнаруженной (белой) до этого момента, иначе поиск уже прошел бы по этому ребру в направлении от *v* к *u* .

Таким образом, (*u, v* \) становится древесным ребром .

Свойства DFS2 Классификация ребер

2) Если поиск сначала исследует { *u, v*} в направлении от *v* к *u*, то { *u, v*} является обратным ребром, поскольку *u* все еще серая в момент первого прохода по ребру.



Приложения бэктрэкинга

Существуют проблемы, которые можно решить только путем полного перебора всех возможных решений .

Одним из способов систематического поиска решения является использование дерева решений, где каждая внутренняя вершина представляет выбор, а каждый лист — возможное решение .

Чтобы найти решение методом отката, сначала постройте последовательность решений в попытке достичь решения, пока это возможно.

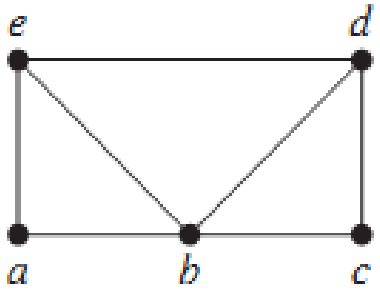
Последовательность решений можно представить в виде пути в дереве решений.

Как только станет известно, что из дальнейшей последовательности выборов не может быть получено никакого решения, **вернитесь к родительской вершине текущей вершины** и работайте над решением с помощью другой серии выборов, если это возможно.

Процедура продолжается до тех пор, пока не будет найдено решение или не будет установлено, что решения не существует .

ПРИМЕР бэктрэкинга. Раскраска графа

Как можно использовать метод бэктрэкинга, чтобы решить, можно ли раскрасить граф с использованием *n* цветов?



ПРИМЕР бэктрэкинга. Раскраска графа

Решение: Эту проблему можно решить с помощью бэктрэкинга следующим образом.

Сначала выберем некоторую вершину *a* и присвоим ей цвет 1 .

Затем выберем вторую вершину *b* и

если *b* не является смежной с *a* , присвоим ей цвет 1.

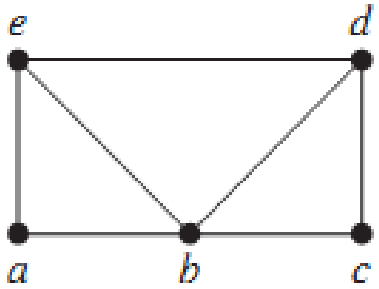
В противном случае присвоим *b цвет 2* .

Затем переходим к третьей вершине *c* .

используем цвет 1 для *c* .

В противном случае используем цвет 2 , если это возможно.

Только если ни цвет 1, ни цвет 2 не могут быть использованы, следует использовать цвет 3, пока можно будет присвоить один из *n* цветов каждой следующей вершине, всегда используем первый допустимый цвет в списке .



ПРИМЕР бэктрэкинга. Раскраска графа

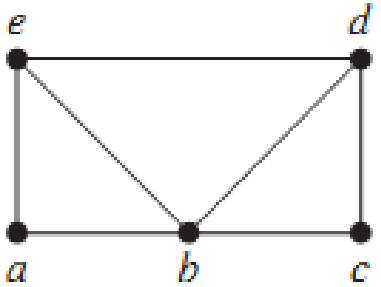
Если достигнута вершина, которую нельзя покрасить ни в один из *n* цветов, вернуться к последнему выполненному присваиванию и изменить цвет последней окрашенной вершины , если это возможно, используя следующий допустимый цвет в списке.

Если изменить эту окраску невозможно, вернуться к предыдущим присваиваниям, делая один шаг назад до тех пор, пока не появится возможность изменить цвет вершины.

Затем продолжаем присваивать цвета следующим вершинам как можно дольше.

Если существует раскраска с использованием *n* цветов, бэктрэкинг найдет ее.

(К сожалению, эта процедура может оказаться крайне неэффективной .)



ПРИМЕР бэктрэкинга. Раскраска графа

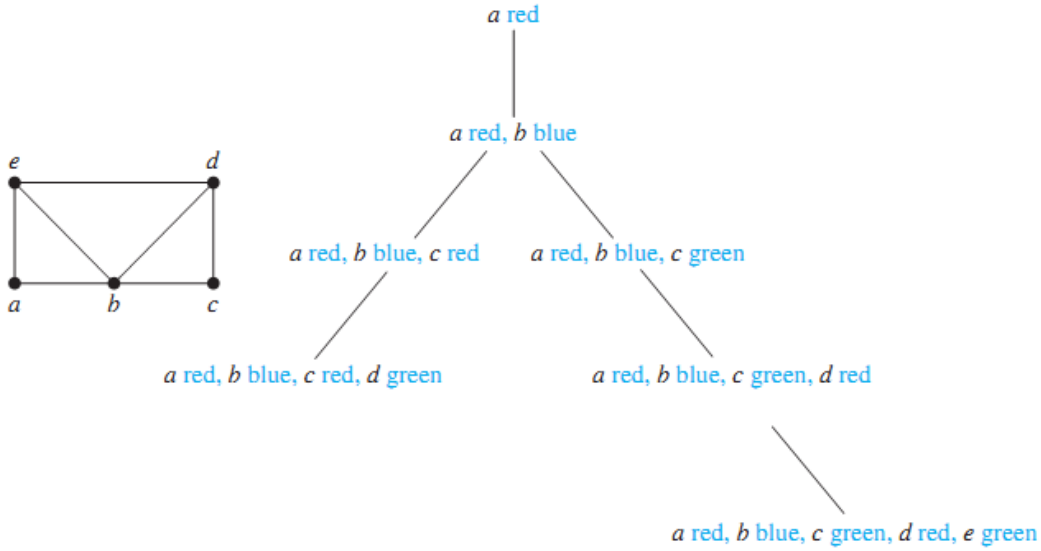
В частности, рассмотрим задачу раскраски графа, показанного ниже, тремя цветами.

Приведенное ниже дерево иллюстрирует, как можно использовать возврат для построения трехцветной раскраски.

В этой процедуре сначала используется красный цвет , затем синий и, наконец, зеленый .

Этот простой пример, очевидно, можно реализовать без откатов.

В этом дереве начальный путь от корня, представляющий собой присвоение красного цвету *a*, ведет к окраске вершины *a в* красным цвет, *b* синий, *c* красный и *d* зеленый.



ПРИМЕР бэктрэкинга. Раскраска графа

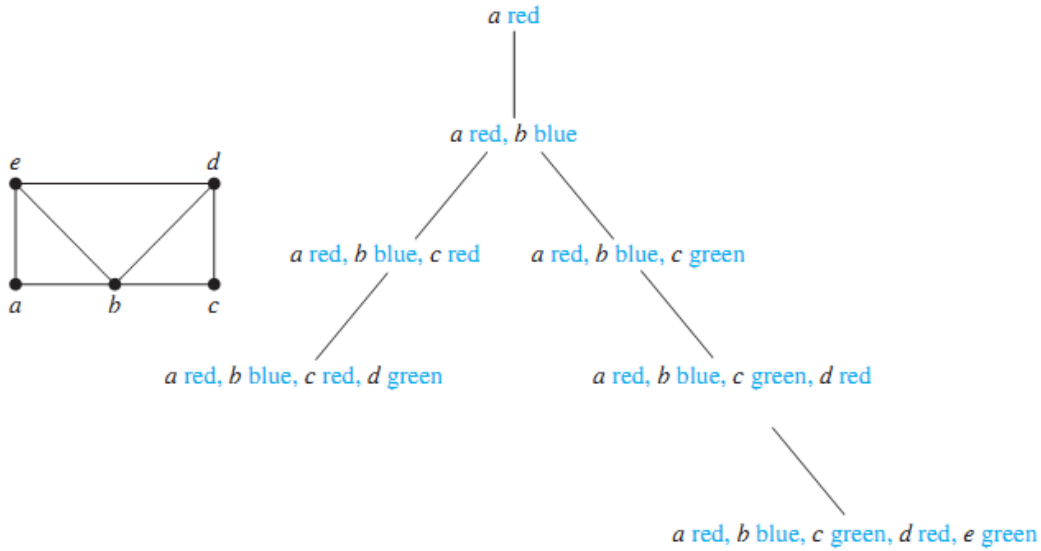
Если *a, b, c* и *d* окрашены так, то невозможно раскрасить *e,* используя любой из трех цветов.

Итак, вернемся к отцу вершины, представляющей эту раскраску.

Поскольку никакой другой цвет не может быть использован для *d*, вернемся еще на один уровень назад.

Затем изменим цвет *c* на зеленый .

Мы получаем раскраску графа, назначая красный цвет *d* и зеленый цвет *e.*



ПРИМЕР бэктрэкинга. Задача про *n* ферзей

Задача об *n* ферзях заключается в том, чтобы разместить *n ферзей на* шахматной доске *размером n* × *n* так, чтобы никакие два ферзя не могли атаковать друг друга.

Как можно использовать бэктрэкинг для решения проблемы *n* ферзей?

Решение : Чтобы решить эту задачу, мы должны найти *n* позиций на шахматной доске *n* × *n* так, чтобы никакие 2 из этих позиций не находились в одной строке, одном столбце или на одной диагонали Мы будем использовать бэктрэкинг для решения задачи *n* ферзей.

Начнем с пустой шахматной доски.

На этапе *k* + 1 мы пытаемся поставить на доску дополнительного ферзя в (*k* + 1)-й столбец, где в первых *k* столбцах уже есть ферзи.

ПРИМЕР бэктрэкинга. Задача про *n* ферзей

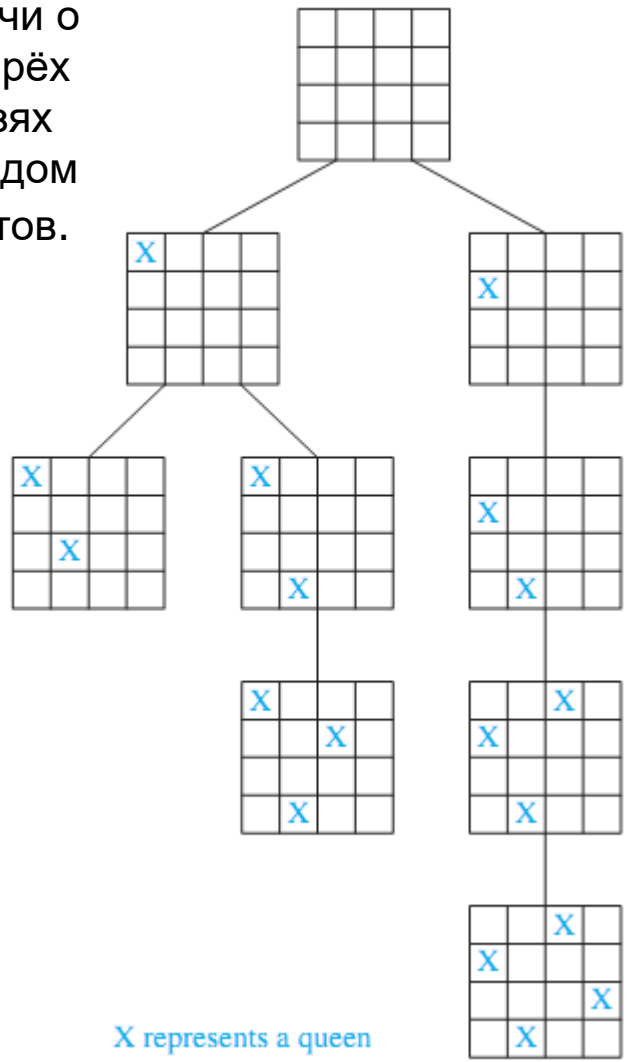
Мы рассматриваем квадраты в (*k* + 1)-м столбце, начиная с с поля в первом ряду, ищем позицию, чтобы разместить этого ферзя так, чтобы он не находился в в том же ряду или на той же диагонали, что и ферзь, уже находящийся на доске.

Если невозможно найти позицию для размещения ферзя в (*k* + 1)-м столбце, вернемся к размещению ферзя в *k* -м столбце и поместим этого ферзя в следующую допустимую строку в этом столбце, если такая строка существует.

Если такой строки нет, вернемся еще на шаг назад.

ПРИМЕР бэктрэкинга. Задача про *n* ферзей

Решение задачи о четырёх ферзях методом откатов.



На рисунке показано решение задачи с четырьмя ферзями методом откатов.

Сначала мы помещаем ферзя в первую строку и столбец.

Затем мы ставим ферзя во второй столбец, 3-ю рстроку.

Однако это делает невозможным размещение ферзя в 3-ем столбце.

Тогда мы возвращаемся назад и ставим ферзя в 4-ю строку второго столбца.

Когда мы это сделаем, мы сможем поместить ферзя во вторую строку 3-го столбца.

Но нет возможности добавить ферзя в четвертый столбец.

Это показывает, что при размещении ферзя в первой строке и столбце решения нет.

Возвращаемся к пустой шахматной доске и ставим ферзя во вторую строку первого столбца, и т.д.

Пример бэктрэкинга. Поиск заданной суммы подмножеств

Дано множество положительных целых чисел x_1, x_2, ..., x_n. Найти подмножество этого множества целых чисел, сумма которого равна *M.*

Как можно использовать откат для решения этой проблемы?

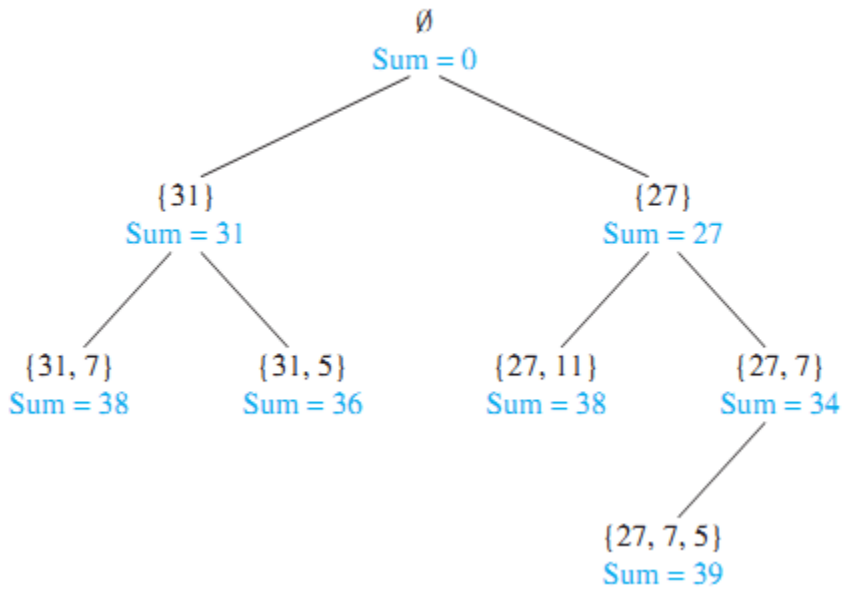
Решение: Начнем с суммы без членов.

Последовательно добавляя члены, увеличиваем сумму.

Целое число включается в последовательность, если при добавлении этого целого числа к сумме она остается $\leq M$.

Если сумма достигнута так, что добавление любого члена > *M*, вернуться назад, отбросив последний член суммы.

Решение методом обратного поиска задачи нахождения подмножества {31, 27, 15, 11, 7, 5} с суммой = 39.



- Ваши вопросы?
- Контакты лектора:
  apanovich_09@mail.ru