Тема 6. Алгоритмы на графах

6.4. Остовные деревья

6.4.3. Минимальное остовное дерево

Для взвешенного графа часто требуется определить остовное дерево (лес) с минимальным общим весом ребер. Остовное дерево (лес), у которого сумма весов всех его ребер минимальна, называется *минимальным остовным деревом*. Рассмотрим два достаточно популярных и эффективных алгоритма построения минимального остовного дерева.

Алгоритм Крускала

Алгоритм Крускала (Kruskal) известен как жадный алгоритм. Идея алгоритма проста и заключается в следующем. На каждом шаге выбирается новое ребро с наименьшим весом, не образующее циклов с ранее выбранными ребрами. Процесс продолжается до тех пор, пока не будет выбрано $|V| - 1$ ребер, образующих остовное дерево.

Вариант реализации алгоритма Крускала, который строит остовное дерево $T = (V, E_T)$ с минимальным общим весом W_T для взвешенного неориентированного графа $G = (V, E)$, $E_T \subseteq E$, представлен алгоритмом 6.5.

$W_T \leftarrow 0$ // сумма весов ребер остовного дерева
$E_T \leftarrow \varnothing$ // множество ребер остовного дерева
$VS \leftarrow \{\{v\} : v \in V\}$
Упорядочить список ребер E по неубыванию весов
while $|VS| > 1$
do
- Выбрать из E ребро (v, u) с наименьшим весом
- Удалить (v, u) из E
- **if** $v \in A, u \in B, A, B \in VS, A \neq B$
- **then**
  - // v и u принадлежат различным подмно-
  - // жествам A и B из VS. Заменить в VS
  - // подмножества A и B на $A \cup B$
  - $VS \leftarrow (VS - \{A, B\}) \cup \{A \cup B\}$
  - $E_T \leftarrow E_T \cup \{(v, u)\}$
  - $W_T \leftarrow W_T + w_{vu}$ // w_{vu} – вес ребра (v, u)

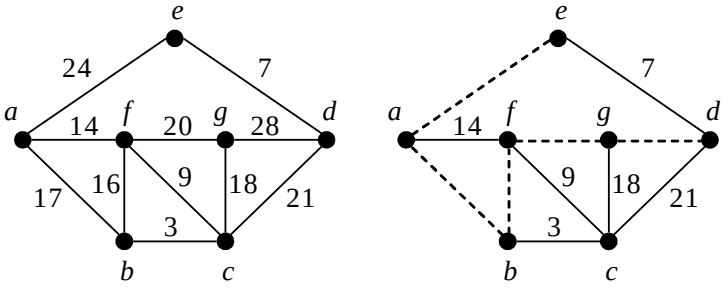
Алгоритм 6.5. Алгоритм Крускала

Основой работы алгоритма является множество VS, которое представляет собой разбиение множества вершин V на непересекающиеся подмножества, т. е. на связные компоненты формируемого остовного дерева, определяемые текущим содержимым множества E_T ребер дерева. Очевидно, что вначале, когда еще ни одно ребро не включено в E_T, VS содержит $|V|$ одноэлементных подмножеств, каждое из которых представляет собой вершину $v \in V$. Ребра выбираются из E в порядке возрастания стоимости и рассматриваются по очереди. Ребро (v, u), анализируемое в условном операторе **if**, добавляется в E_T тогда и только тогда, когда v и u принадлежат разным подмножествам A и B из VS, в противном случае его добавление вызвало бы появление цикла в соответствующей связной компоненте. Добавление ребра в E_T означает, что оно соединяет две связные компоненты в одну. Поэтому в VS необходимо заменить подмножества A и B на их объединение $A \cup B$. Алгоритм завершает работу, когда VS будет содержать точно одно множество, содержащее все вершины из V. Это означает, что все вершины входят в одну связную компоненту, не имеющую цикла, т. е. завершено построение остовного дерева. Следует обратить внимание на то, что к моменту завершения построения остовного дерева могут остаться неисследованные ребра графа.

Для иллюстрации работы алгоритма рассмотрим построение минимального остовного дерева для неориентированного взвешенного графа (рис. 6.4, *а*). Числа рядом с ребрами указывают их веса. Перечислим ребра в порядке возрастания их весов

Ребро:	(b,c)	(d,e)	(c,f)	(a,f)	(b,f)	(a,b)	(c,g)	(f,g)	(c,d)	(a,e)	(d,g)
Вес:	3	7	9	14	16	17	18	20	21	24	28

Полученное минимальное остовное дерево с минимальным общим весом ребер $W_T = 72$ представлено на рис. 6.4, *б*; последовательность шагов построения остовного дерева – на рис. 6.4, *в*.



а *б*

Ребро	Вес	Действие	Множества в *VS*
			{*a*}, {*b*}, {*c*}, {*d*}, {*e*}, {*f*}, {*g*}
(*b*, *c*)	3	Добавить	{*a*}, {*b*, *c*}, {*d*}, {*e*}, {*f*}, {*g*}
(*d*, *e*)	7	Добавить	{*a*}, {*b*, *c*}, {*d*, *e*}, {*f*}, {*g*}
(*c*, *f*)	9	Добавить	{*a*}, {*b*, *c*, *f*}, {*d*, *e*}, {*g*}
(*a*, *f*)	14	Добавить	{*a*, *b*, *c*, *f*}, {*d*, *e*}, {*g*}
(*b*, *f*)	16	Отвергнуть	
(*a*, *b*)	17	Отвергнуть	
(*c*, *g*)	18	Добавить	{*a*, *b*, *c*, *f*, *g*}, {*d*, *e*}
(*f*, *g*)	20	Отвергнуть	
(*c*, *d*)	21	Добавить	{*a*, *b*, *c*, *d*, *e*, *f*, *g*}

в

Рис. 6.4. Построение остовного дерева алгоритмом Крускала:
а – исходный неориентированный граф; *б* – минимальное остовное дерево;
в – последовательность шагов построения минимального остовного дерева

Как видно из примера, ребра (a, e) и (d, g) алгоритмом не исследованы, так как включением в E_T дуги (c, d) завершилось построение остовного дерева. Данный факт позволяет сделать некоторые рекомендации по выбору методов сортировки ребер.

Во-первых, поскольку сортировка существенно влияет на эффективность алгоритма в целом, необходимо применять наиболее эффективные методы сортировки. Во-вторых, поскольку в общем случае не требуется полная сортировка ребер, лучше использовать сортировку, основанную на выборе, которая позволяет остановить сортировку после выполнения k шагов, расположив первые k элементов в их окончательные позиции.

С рассмотренных позиций наиболее подходящей является пирамидальная сортировка. При этом необходимо учесть, что реально ребра не сортируются, а хранятся в виде пирамиды, корнем которой является ребро с наименьшим весом. Другими словами, список ребер графа представляется как очередь с приоритетами (приоритетом является вес ребра), одним из идеальных способов реализации которой является пирамида. Выбор ребра с наименьшей стоимостью (корня пирамиды) и его исключение из очереди предполагает восстановление пирамиды. Возможно применение и других структур данных для эффективной организации очередей с приоритетами, например 2–3-деревья.

Другим существенным фактором, влияющим на сложность алгоритма, является способ представления множества *VS* и способ слияния его подмножеств. Можно воспользоваться следующим методом. Каждое подмножество множества *VS* представляется ориентированным деревом, вершинам которого сопоставлены элементы подмножества. Из каждой вершины существует путь до корня (имеется указатель отца). Корень идентифицирует подмножество, т. е. элемент, сопоставленный корню, есть имя подмножества. Каждая вершина содержит информацию о размере поддерева (число вершин в поддереве), корнем которого она является. Таким образом, разбиение *VS* представляется в виде леса, состоящего из деревьев, соответствующих подмножествам из *VS*. Например, лес для разбиения $VS = \{\{a\},\{b, c, d, e\},\{f, g\}\}$ показан на рис. 6.5, *а*; именами подмножеств являются соответственно элементы *a*, *c* и *f*.

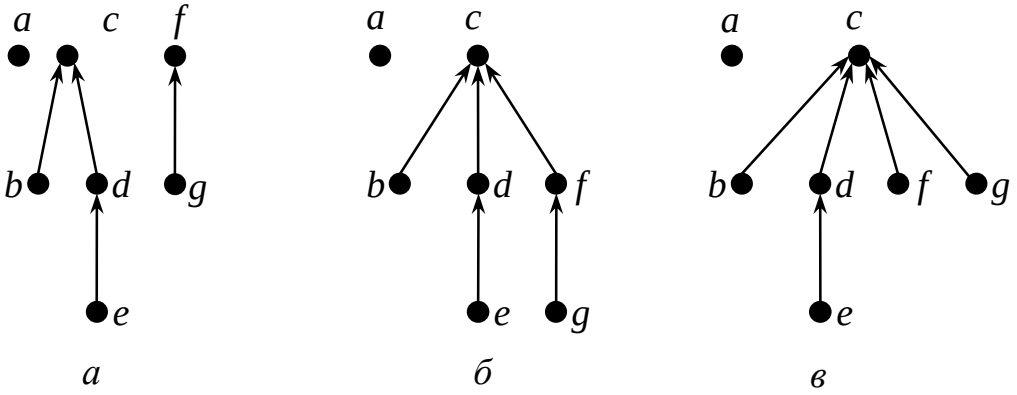


Рис. 6.5. Представление множества *VS*:
а – *VS* = {{*a*},{*b*, *c*, *d*, *e*},{*f*, *g*}}; *б* – *VS* = {{*a*},{*b*, *c*, *d*, *e*, *f*, *g*}};
в – *VS* = {{*a*},{*b*, *c*, *d*, *e*, *f*, *g*}} после сжатия пути

Объединение подмножеств A и B происходит добавлением дуги от корня дерева, представляющего одно подмножество, к корню дерева, представляющего другое подмножество. Для большей эффективности лучше использовать *объединение с балансировкой*. Балансировка заключается в назначении корня большего по размеру дерева отцом корня меньшего дерева, т. е. имя большего подмножества становится именем подмножества, получившегося в результате объединения. Тогда высота каждого дерева будет не больше логарифма от числа его вершин (а следовательно, и от числа дуг). Результат объединения множеств $\{b, c, d, e\}$ и $\{f, g\}$ представлен на рис. 6.5, *б*; именем получившегося подмножества будет элемент c.

Операция поиска элемента *x* состоит в продвижении по указателям отцов от *x* до корня (до имени подмножества). Для повышения эффективности поиска можно использовать операцию *сжатия пути*, которая заключается в следующем. После завершения выполнения поиска элемента *x* все вершины на пути между *x* и корнем (включая и вершину *x*) становятся сыновьями корня. Например, если выполнить операцию поиска элемента *g* на лесе, показанном на рис. 6.5, *б*, то в качестве имени подмножества, содержащего *g*, определится элемент *c*. Сжатие пути приведет к изменению леса, который будет иметь вид, как на рис. 6.5, *в*. Сжатие пути незначительно увеличивает сложность поиска, но дает существенный выигрыш во времени при достаточно большом числе операций поиска.

Пусть каждый элемент *x* множества представляется узлом, состоящим из поля *info*, в котором хранится элемент *x* множества, поля связи *father*, указывающего на отца узла, и поля *size*, в котором содержится число узлов в поддереве с корнем *x*. Кроме того, для обеспечения прямого доступа к элементам множества (т. е. к узлам леса) имеется набор внешних указателей l_x, по одному для каждого элемента *x* множества. Тогда функцию поиска со сжатием пути *FIND* (*x*), которая выдает имя подмножества, содержащего *x*, можно реализовать алгоритмом 6.6, а процедуру объединения *UNION* (*x*, *y*), образующую новое подмножество, содержащее все элементы подмножеств с именами *x* и *y*, – алгоритмом 6.7.

function $FIND(x)$

// x – узел леса, записанный в ячейку l_x

$S \leftarrow \varnothing$ // стек S пуст

$t \leftarrow l_x$

while $father(t) \neq \Lambda$ **do** $\begin{cases} S \Leftarrow t \\ t \leftarrow father(t) \end{cases}$

while $S \neq \varnothing$ **do** $\begin{cases} v \Leftarrow S \\ father(v) \leftarrow t \\ size(v) \leftarrow 1 \end{cases}$

$FIND \leftarrow t$

return

Алгоритм 6.6. Функция поиска со сжатием пути

procedure $UNION(x, y)$

// x и y – корни деревьев, записанные в ячейках l_x и l_y

if $size(l_x) < size(l_y)$

then $\begin{cases} father(l_x) \leftarrow l_y \\ size(l_y) \leftarrow size(l_x) + size(l_y) \end{cases}$

else $\begin{cases} father(l_y) \leftarrow l_x \\ size(l_x) \leftarrow size(l_y) + size(l_x) \end{cases}$

return

Алгоритм 6.7. Процедура объединения подмножеств с балансировкой

Таким образом, в алгоритме Крускала после выбора ребра (v, u) необходимо выполнить следующую последовательность операций:

$x \leftarrow FIND(v)$
$y \leftarrow FIND(u)$
if $x \neq y$ **then** $UNION(x, y)$.

Эффективность алгоритма Крускала зависит от организации выбора ребра с наименьшим весом, способа представления разбиения *VS* и способа объединения подмножеств. Построение очереди с приоритетами на основе пирамиды требует $O(|E|)$ операций, восстановление пирамиды после выбора и исключения ребра с наименьшей стоимостью – $O(\log|E|)$ операций. Поскольку число операций выбора и исключения ребра из очереди пропорционально $|E|$, то общее число операций составит $O(|E|\log|E|)$. Нахождение подмножеств A и B требует, как уже отмечалось, $O(\log|E|)$ операций. Объединение подмножеств занимает постоянное время, не зависящее от их размеров. Общее число нахождения подмножеств и их объединений пропорционально числу ребер, т. е. требуется $O(|E|\log|E|)$ операций. Первоначальное разбиение *VS* на одноэлементные подмножества занимает время $O(|V|)$. Таким образом, временная сложность алгоритма Крускала составляет $O(|E|\log|E|)$.

Алгоритм Дейкстры – Прима

Алгоритм Дейкстры – Прима (Dijkstra, Prim) известен как алгоритм ближайшего соседа. Этот алгоритм, в отличие от алгоритма Крускала, не требует решать задачи ни сортировки ребер, ни проверки на цикличность на каждом шаге. Алгоритм начинает построение минимального остовного дерева с некоторой произвольной вершины v графа $G = (V, E)$. Выбирается ребро (v, w), инцидентное вершине v, с наименьшим весом и включается в дерево. Затем среди инцидентных либо v, либо w выбирается ребро с наименьшим весом и включается в частично построенное дерево. В результате в дерево добавляется новая вершина, например u. Ищется наименьшее ребро, соединяющее v, w или u с некоторой другой вершиной графа. Процесс продолжается до тех пор, пока все вершины из G не будут включены в дерево, т. е. пока дерево не станет остовным.

Очевидно, что основным фактором, влияющим на эффективность алгоритма, является поиск ребер с наименьшим весом, инцидентных всем вершинам, включенным в частично построенное дерево. Худшим является случай, когда G – полный граф, т. е. каждая пара вершин графа соединена ребром. В этом случае для поиска ближайшего соседа на каждом шаге необходимо сделать максимальное число сравнений. Для выбора первого ребра нужно сравнить веса всех $|V|-1$ ребер, инцидентных вершине v, т. е. требуется $|V|-2$ сравнений. Для выбора второго ребра ищется наименьшее из возможных $2(|V|-2)$ ребер, инцидентных v или w, что требует $2(|V|-2)-1$ сравнений. Таким образом, для выбора i-го ребра нужно $i(|V|-i)-1$ сравнений, а общее число сравнений для построения остовного дерева будет составлять

$$\sum_{i=1}^{|V|-1}[i(|V|-1)-1]=O(|V|^3).$$

Для уменьшения числа сравнений можно каждой вершине v, еще не принадлежащей дереву, сопоставить указатель вершины дерева, ближайшей к v. Имея такую информацию, i-ю добавляемую вершину можно найти за $|V| - i$ сравнений при $i \geq 2$. Первая вершина выбирается произвольно и не требует сравнений. После добавления к дереву i-й вершины информацию о ближайших вершинах можно скорректировать за $|V| - i$ операций, сравнивая для каждой вершины v (из $|V| - i$ вершин), не принадлежащей дереву, расстояния от нее до только что добавленной и до той, которая перед этим была в дереве ближайшей к ней. Поэтому для $i \geq 2$ добавление к дереву i-й вершины требует $2(|V| - i)$ сравнений, а общее число сравнений

$$\sum_{i=1}^{|V|} 2(|V| - i) = (|V| - 1)(|V| - 2) = O(|V|^2).$$

Процедура построения минимального остовного дерева $T = (V, E_T)$ с общим весом W_T для графа $G = (V, E)$, заданного матрицей весов $W = [w_{ij}]$, использующий рассмотренную технику, представлена алгоритмом 6.8 (в матрице W веса всех несуществующих ребер равны ∞).

Выбрать произвольно v

for $u \in V$ **do** $\begin{cases} dist(u) \leftarrow w_{uv} \\ near(u) \leftarrow v \end{cases}$

$V_T \leftarrow \{v\}$ // множество вершин остовного дерева

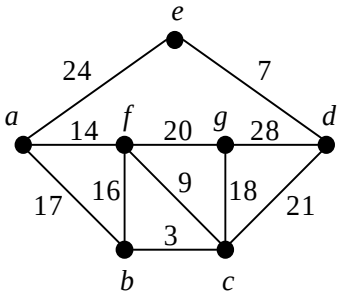
$E_T \leftarrow \varnothing$ // множество ребер остовного дерева

$W_T \leftarrow 0$ // сумма весов ребер остовного дерева

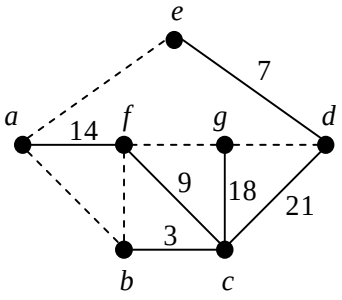
while $V_T \neq V$ **do** $\begin{cases} \text{Выбрать из } V - V_T \text{ вершину } v \\ \quad \text{с наименьшим значением } dist(v) \\ V_T \leftarrow V_T \cup \{v\} \\ E_T \leftarrow E_T \cup \{(v, near(v))\} \\ W_T \leftarrow W_T + dist(v) \\ \textbf{for } x \in V - V_T \textbf{ do} \\ \quad \textbf{if } dist(x) > w_{vx} \textbf{ then } \begin{cases} dist(x) \leftarrow w_{vx} \\ near(x) \leftarrow v \end{cases} \end{cases}$

Алгоритм 6.8. Алгоритм Дейкстры – Прима

В алгоритме массивы $dist(u)$ для расстояний (весов) и $near(u)$ для названий вершин используются для записи текущей ближайшей вершины в дереве для каждой из вершин, еще не попавших в него. На i-м шаге цикла **while** множество $V - V_T$ содержит $|V| - i$ элементов. Поэтому поиск вершины v с наименьшим значением $dist(v)$ требует $|V| - i - 1$ сравнений. Операторы присваивания для V_T, E_T и W_T требуют фиксированного числа операций. Цикл **for** выполняет $|V| - i - 1$ сравнений «$dist(x) > w_{vx}$». Поэтому общее число сравнений равно $O(|V|^2)$.



а



б

Выбрать произвольно v

for $u \in V$ **do** $\begin{cases} dist(u) \leftarrow w_{uv} \\ near(u) \leftarrow v \end{cases}$

$V_T \leftarrow \{v\}$ // множество вершин остовного дерева
$E_T \leftarrow \varnothing$ // множество ребер остовного дерева
$W_T \leftarrow 0$ // сумма весов ребер остовного дерева

while $V_T \neq V$ **do** $\begin{cases} \text{Выбрать из } V - V_T \text{ вершину } v \\ \quad \text{с наименьшим значением } dist(v) \\ V_T \leftarrow V_T \cup \{v\} \\ E_T \leftarrow E_T \cup \{(v, near(v))\} \\ W_T \leftarrow W_T + dist(v) \\ \textbf{for } x \in V - V_T \textbf{ do} \\ \quad \textbf{if } dist(x) > w_{vx} \textbf{ then } \begin{cases} dist(x) \leftarrow w_{vx} \\ near(x) \leftarrow v \end{cases} \end{cases}$

v	*dist*(*v*)							*near*(*v*)							V_T	W
	a	*b*	*c*	*d*	*e*	*f*	*g*	*a*	*b*	*c*	*d*	*e*	*f*	*g*		
a	–	17	–	–	24	**14**	–	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	0
f	–	16	**9**	–	24	**14**	20	*a*	*b*	*c*	*a*	*a*	*a*	*g*	*a,f*	14
c	–	3	**9**	21	24	**14**	18	*a*	*c*	*c*	*c*	*a*	*a*	*c*	*a,f,c*	23
b	–	**3**	**9**	21	24	**14**	**18**	*a*	*c*	*c*	*c*	*a*	*a*	*c*	*a,f,c,b*	26
g	–	**3**	**9**	**21**	24	**14**	**18**	*a*	*c*	*c*	*c*	*a*	*a*	*c*	*a,f,c,b,*g	44
d	–	**3**	**9**	**21**	7	**14**	**18**	*a*	*c*	*c*	*c*	*d*	*a*	*c*	*a,f,c,b,*g*,d*	65
e	–	**3**	**9**	**21**	**7**	**14**	**18**	*a*	*c*	*c*	*c*	*d*	*a*	*c*	*a,f,c,b,*g*,d,e*	72

E_T = {(*a*,*f*), (*f*,*c*), (*c*,*b*), (*g*,*c*), (*d*,*c*), (*e*,*d*)}