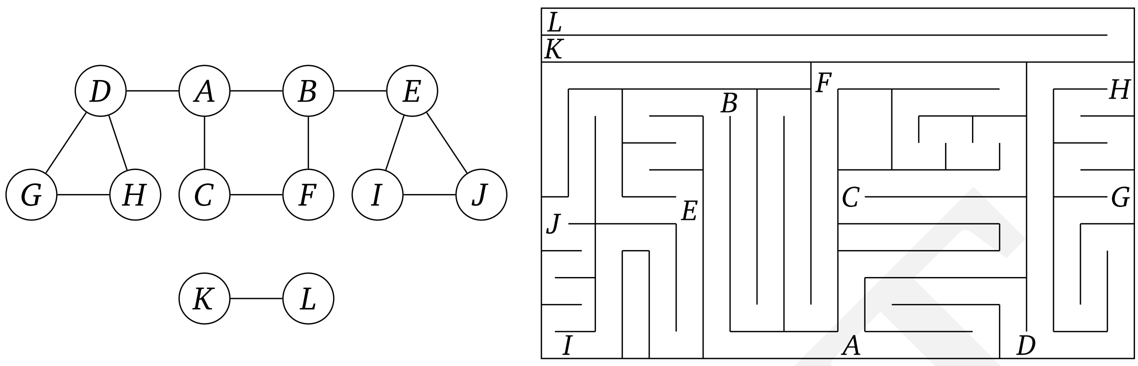
Рис. 3.2. Лабиринт и его граф.



мы добавляем его в стек, а возвращаясь обратно, забираем), но в нашем алгоритме (рис. 3.3) мы вместо явного стека используем рекурсию.[1]

В нашем алгоритме указано также место для процедур Previsit и Postvisit (вызываемых до и после обработки вершины); мы увидим дальше, зачем это может быть полезно.

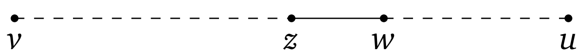
Рис. 3.3. Нахождение всех вершин, достижимых из данной.

процедура Explore(v)

{Вход: вершина v графа $G=(V,E)$.}
{Выход: visited$[u]=$true для всех вершин u, достижимых из v.}

visited$[v]\leftarrow$true
Previsit(v)
для каждого ребра $(v,u)\in E$:
  если visited$[u]=$false: Explore(u)
Postvisit(v)

А пока нужно убедиться, что процедура Explore корректна. Ясно, что она обходит только достижимые из v вершины, поскольку на каждом шаге она переходит от вершины к её соседу. Но обходит ли она *все* такие вершины? Допустим, что нашлась достижимая из v вершина u, до которой Explore почему-то не добралась. Рассмотрим тогда какой-нибудь путь из v в u. Пусть z — последняя вершина этого пути, которую посетила процедура Explore (сама вершина v, если других нет), а w — следующая сразу за z вершина на этом пути:



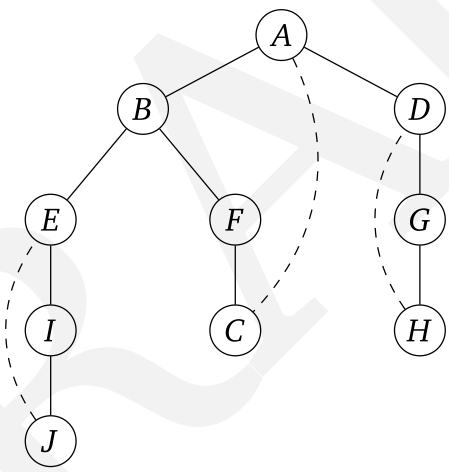
[1]Этот алгоритм (как и многие другие) мы излагаем для ориентированных графов; чтобы использовать его для неориентированных, надо представить неориентированное ребро $\{x,y\}$ парой ориентированных (x,y) и (y,x).

Выходит, что вершину z процедура посетила, а вершину w — нет. Но так быть не может: ведь состоявшийся вызов Explore(z) должен был перебрать всех соседей z.

(Уточнение: начальный вызов процедуры Explore(v) предполагает, что $\texttt{visited}[u] = \texttt{false}$ для всех вершин u; в противном случае она обработает лишь часть графа, где это было так.)

На рис. 3.4 показан вызов Explore для рассмотренного ранее графа и вершины A. Считаем, что соседи вершины перебираются в алфавитном порядке. Сплошными нарисованы рёбра, которые ведут в ранее не встречавшиеся вершины. Например, когда процедура Explore находилась в вершине B, она прошла по ребру B–E, и, поскольку в E она до этого не бывала, был произведён вызов Explore для E. Сплошные рёбра образуют дерево (связный граф без циклов) и поэтому называются *древесными рёбрами* (tree edges). Пунктирные же рёбра ведут в вершины, которые уже встречались. Такие рёбра называются *обратными* (back edges).

Рис. 3.4. Результат вызова Explore(A) для графа с рис. 3.2.



3.2.2. Поиск в глубину

Процедура Explore обходит все вершины, достижимые из данной. Для обхода всего графа алгоритм *поиска в глубину* (рис. 3.5) последовательно вызывает Explore для всех вершин (пропуская уже посещённые).

Рис. 3.5. Поиск в глубину.

процедура DFS(G)

для всех вершин $v \in V$:
  $\texttt{visited}[v] \leftarrow \texttt{false}$

для всех вершин $v \in V$:
  если $\texttt{visited}[v] = \texttt{false}$: Explore($v$)

Сразу же отметим, что процедура Explore вызывается для каждой вершины ровно один раз благодаря массиву `visited` (пометки в лабиринте). Сколько времени уходит на обработку вершины? Она включает в себя:

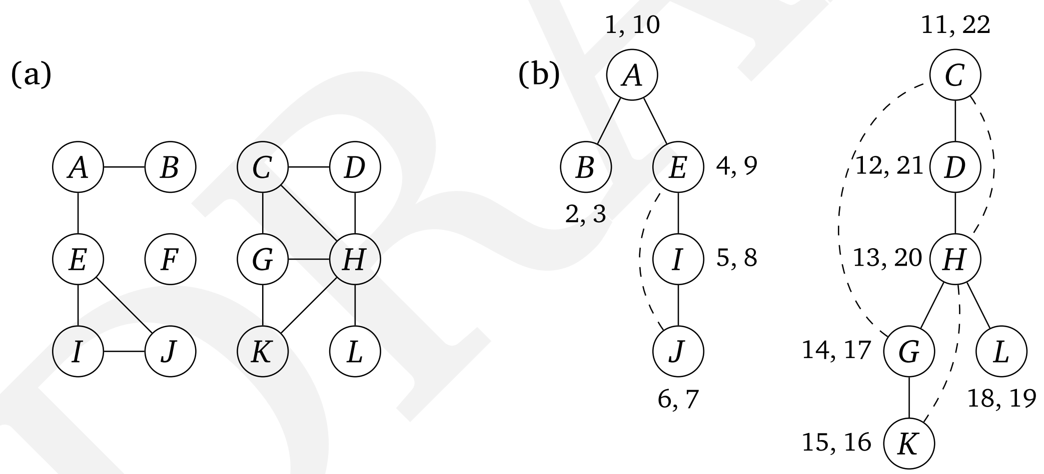
1) $O(1)$-операции: пометка вершины, а также вызовы Previsit и Postvisit (мы не учитываем время внутри этих вызовов);

2) перебор соседей.

Общее время зависит от вершины, но можно оценить количество операций для всех вершин вместе. Общее количество операций на шаге 1 есть $O(|V|)$. На шаге 2 (будем говорить о неориентированном графе) каждое ребро $\{x, y\} \in E$ просматривается ровно *два раза*: при вызовах Explore(x) и Explore(y). Общее время работы на шаге 2, таким образом, есть $O(|E|)$. В целом время работы поиска в глубину есть $O(|V| + |E|)$, то есть линейно. За меньшее время мы не успеем даже прочесть все вершины и рёбра!

На рис. 3.6 показан поиск в глубину на графе с двенадцатью вершинами (не обращайте пока внимания на пары чисел). Опять считаем, что соседи перебираются в алфавитном порядке. Во внешнем цикле процедура Explore вызывается трижды — для вершин A, C и F. Результатом является *лес* (forest) из трёх деревьев с корнями в этих вершинах.

Рис. 3.6. (a) Граф на двенадцати вершинах. (b) Лес, построенный поиском в глубину.



3.2.3. Связные неориентированные графы

Неориентированный граф называется *связным* (connected), если любые две его вершины соединены путём (по рёбрам). Граф на рис. 3.6 не связен: например, нет пути из A в K. Этот граф можно разбить на три непересекающихся связных подмножества:

$$\{A, B, E, I, J\}, \ \{C, D, G, H, K, L\}, \ \{F\}.$$

Они называются *компонентами связности* (connected components). Каждое из них образует связный подграф, а друг с другом они не соединены. Процедура EXPLORE обходит как раз компоненту связности той вершины, для которой она была вызвана. При каждом вызове EXPLORE во внешнем цикле алгоритма DFS обходится новая компонента связности. Таким образом, с помощью поиска в глубину легко проверить связность графа. Более того, можно для каждой вершины v найти номер её компоненты связности $\mathtt{ccnum}[v]$ (в порядке обнаружения). Для этого нужно завести переменную `cc`, изначально равную нулю, и увеличивать её на единицу каждый раз, когда EXPLORE вызывается во внешнем цикле, а процедуру PREVISIT сделать такой:

процедура PREVISIT(v)

$\mathtt{ccnum}[v] \leftarrow \mathtt{cc}$

3.2.4. Время начала и конца обработки вершины

Как мы видим, поиск в глубину позволяет за линейное время определить, связен ли граф. И это далеко не всё — сейчас мы рассмотрим приём, который лежит в основе многих других применений поиска в глубину. Будем записывать время начала обработки каждой вершины (вызов PREVISIT) и время конца обработки (POSTVISIT). Для нашего примера эти числа (для всех 24 таких событий) показаны на рис. 3.6. Например, в момент времени 5 началась обработка вершины I, а в момент времени 21 закончилась обработка вершины D.

Чтобы сохранить эту информацию при поиске в глубину, заведём счётчик `clock`, изначально равный 1, и напишем так:

процедура PREVISIT(v)

$\mathtt{pre}[v] \leftarrow \mathtt{clock}$
$\mathtt{clock} \leftarrow \mathtt{clock} + 1$

процедура POSTVISIT(v)

$\mathtt{post}[v] \leftarrow \mathtt{clock}$
$\mathtt{clock} \leftarrow \mathtt{clock} + 1$

Вот простое (но важное) свойство этих чисел:

Свойство. *Для любых двух вершин u и v либо отрезки $[\mathtt{pre}(u), \mathtt{post}(u)]$ и $[\mathtt{pre}(v), \mathtt{post}(v)]$ не пересекаются, либо один содержится в другом.*

В самом деле, $[\mathtt{pre}(u), \mathtt{post}(u)]$ — это промежуток времени, в течение которого вершина u была в стеке, и если вершина v была помещена в стек, когда там уже лежала вершина u, то v будет вынута раньше u.

Таким образом, рассмотренные отрезки отражают структуру дерева, построенного при поиске в глубину. Это особенно полезно для ориентированных графов, к которым мы и переходим.

3.3. Поиск в глубину в ориентированных графах

3.3.1. Типы рёбер

Рассмотренный нами алгоритм поиска в глубину может быть использован и для ориентированных графов (в процедуре EXPLORE надо перебирать выходя-