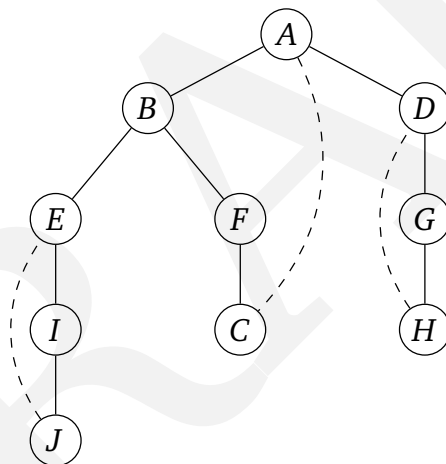


Выходит, что вершину z процедура посетила, а вершину w — нет. Но так быть не может: ведь состоявшийся вызов $\text{EXPLORE}(z)$ должен был перебрать всех соседей z .

(Уточнение: начальный вызов процедуры $\text{EXPLORE}(v)$ предполагает, что $\text{visited}[u] = \text{false}$ 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 $\text{EXPLORE}(A)$ 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 $\text{DFS}(G)$

для всех вершин $v \in V$:

$\text{visited}[v] \leftarrow \text{false}$

для всех вершин $v \in V$:

если $\text{visited}[v] = \text{false}$: $\text{EXPLORE}(v)$

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:

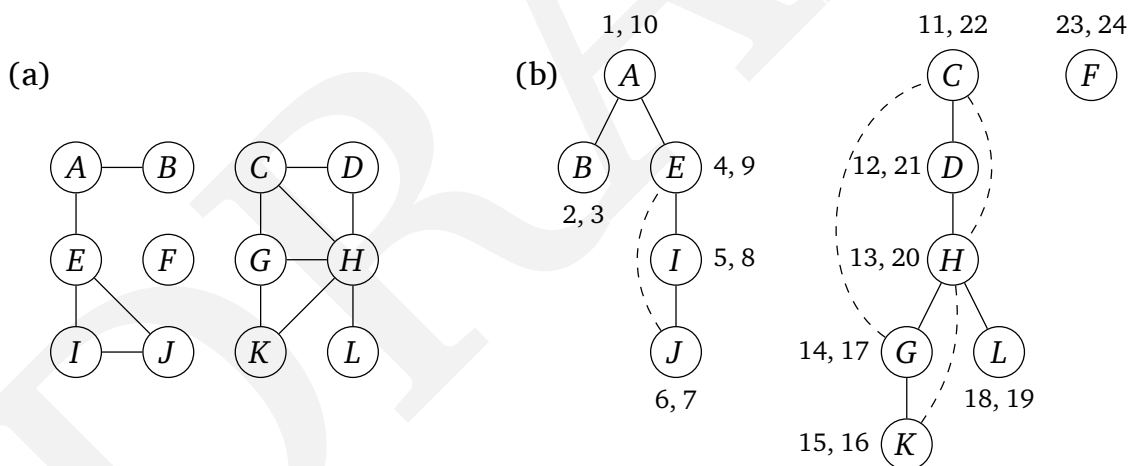
1) $O(1)$ -операции: пометка вершины, а также вызовы PREVISIT и POSTVISIT (мы не учитываем время внутри этих вызовов);

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 обращайтесь 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 связан: 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. Процеду-

ра 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 $ccnum[v]$ (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( $v$ )  
   $ccnum[v] \leftarrow cc$ 
```

3.2.4. Время начала и конца обработки вершины

Как мы видим, поиск в глубину позволяет за линейное время определить, связан 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( $v$ )  
   $pre[v] \leftarrow clock$   
   $clock \leftarrow clock + 1$   
процедура POSTVISIT( $v$ )  
   $post[v] \leftarrow clock$   
   $clock \leftarrow clock + 1$ 
```

Вот простое (но важное) свойство этих чисел:

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

В самом деле, $[pre(u), post(u)]$ — 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-