一、 时间复杂度

(a) 将以下函数按照增长速度排序：

$$f_1(n) = n^{\pi} \qquad f_2(n) = \pi^n \qquad f_3(n) = \binom{n}{5} \qquad f_4(n) = \sqrt{2^{\sqrt{n}}}$$

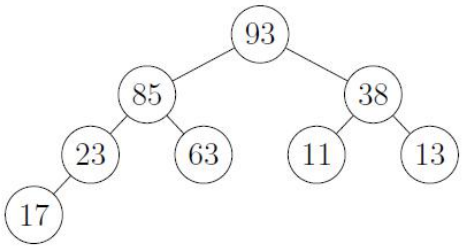
$$f_5(n) = \binom{n}{n-4} \qquad f_6(n) = 2^{\log^4 n} \qquad f_7(n) = n^{5(\log n)^2} \qquad f_8(n) = n^4\binom{n}{4}$$

(b) 求解以下分治算法分析是产生的递归关系：

$$T(n) = T(\tfrac{n}{3}) + T(\tfrac{2n}{3}) + \Theta(n).$$

二、 堆算法

考虑以下堆：



说明这个堆如何使用数组表达。如果这个堆里的最大元素被删除，剩下的堆的数组表达是什么？

三、 最长交替子序列

我们称一个子序列$y_1,y_2,\ldots,y_n$为交替序列，如果每个相邻的y_i,y_{i+1},y_{i+2}都满足$y_i < y_{i+1} > y_{i+2}$或$y_i > y_{i+1} < y_{i+2}$。也就是说，如果$y_i < y_{i+1}$，则$y_{i+1} > y_{i+2}$，反之亦然。给定一个序列$x_1,x_2,\ldots x_n$，我们想要找到这个序列的最长的交替子序列。

例如，如果初始序列是：

$$x_1 = 13, x_2 = 93, x_3 = 86, x_4 = 50, x_5 = 63, x_6 = 4$$

这个序列的最长交替子序列的长度为 5，由 13, 93, 50, 63, 4 组成。

使用动态规划设计寻找最长交替子序列的算法。我们定义子问题$DP(i,b)$，其中$1 \le i \le n$，b 是一个布尔值。如果 b 为真，则 $DP(i,b)$代表最长的终止于x_i的交替子序列，其中最后一步是上升的。如果 b 为假，$DP(i,b)$代表最长的终止于x_i的交替子序列，其中最后一步是下降的。如果子序列的长度为 1，则我们定义它既是上升的也是下降的。例如 $DP(5,\text{True}) = 4$，因为子序列x_1,x_2,x_4,x_5的长度为 4，并且最后一步上升。$DP(5,\text{False}) = 3$，因为子序列x_1,x_2,x_5的长度为 3，并且最后一步下降。

(a) 对于以上序列，计算 $DP(i,b)$，对于每个 $1 \le i \le 6$ 和 $b \in \{\text{True},\text{False}\}$，写入以下表中：

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
$b = \text{True}$						
$b = \text{False}$						

(b) 写出 $DP(i,b)$的递归关系。

(c) 写出 $DP(i,b)$递归关系的基本情况（$i = 1$）。

(d) 如果采用自底向上的计算方式，给出一个合适的计算顺序。

(e) 如果计算了所有的 $DP(i,b)$，如何计算最长交替子序列的长度？

(f) 将以上步骤放到一起，给出计算最长交替子序列长度的算法，使用伪代码表示。分析算法的时间复杂度。

四、 括号小游戏

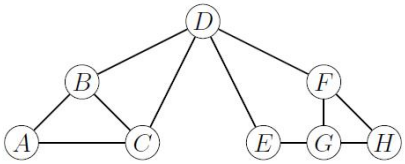
给定一个不带括号的，由加法和乘法组成的表达式，找出如何添加括号，使得表达式的取值最大。

例如，如果提供的表达式是 $6+0\cdot 6$，则应该添加括号为$(6+0)\cdot 6$，结果为 36，而不是 $6+(0\cdot 6)$，结果为 6。再比如，如果提供的表达式是 $0.1\cdot 0.1+0.1$，则应该添加括号为 $(0.1\cdot 0.1)+0.1$，结果为 0.11，而不是 $0.1\cdot(0.1+0.1)$，结果为 0.02。

使用动态规划设计多项式时间的算法，在给定表达式之后，找到最好的添加括号的方法。假设输入的格式为$x_0,o_0,x_1,o_1,\ldots o_{n-1},x_n$，其中每个$x_i$是一个数字，每个$o_i$是加号或乘号。分析算法的时间复杂度。

五、 图的遍历

考虑以下图：



假设该图使用邻接列表(adjacency list)表示，每个节点的相邻节点按照字母顺序排序（例如，D 的相邻节点表示为$[B,C,E,F]$）。

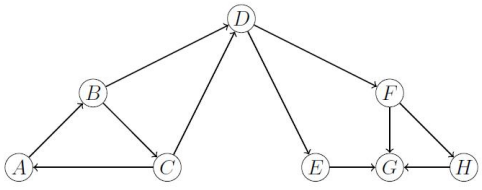
(a) 假设我们使用广度优先搜索(breadth-first search)寻找从 A 到 H 的路径，写出该算法得到的路径。

(b) 假设我们使用深度优先搜索(depth-first search)寻找从 A 到 H 的路径，写出该算法得到的路径。

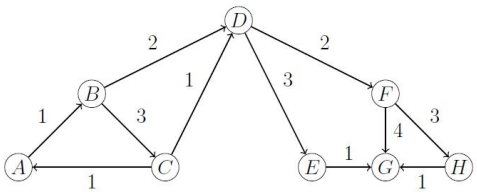
(c) 假设我们从 A 开始采用深度优先搜索，将每条边标记为树边(tree edge, T)、后向边(back edge, B)、前向边(forward edge, F)、和横向边(cross edge, C)。回顾这些概念的定义：一条边(u,v)是

- 树边如果(u,v)在深度优先搜索产生的树中。
- 后向边如果 v 在深度优先树中是 u 的祖先。
- 前向边如果 v 在深度优先数中是 u 的后代。
- 横向边如果以上都不成立。

(d) 重复问题(c)，但使用以下的有向图：



(e) 现在假设图的每条边有一个距离如下：



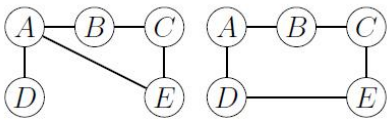
使用 Dijkstra 算法寻找 A 到 H 的最短路径，给出 Dijkstra 算法执行中顶点从优先队列中移出的顺序。

六、 婚礼计划

在婚礼上，你需要为到来的客人安排座位。

(a) 假设你知道客人之间谁互相认识。如果 x 认识 y，则 y 必然认识 x。你需要将客人安排到不同的桌子上，使得每个桌子的客人或者互相认识，或者通过同一个桌子的人认识。例如，如果 x 认识 y，并且 y 认识 z，则 x,y,z 可以坐在同一个桌子上。设计高效的算法，在给定客人之间谁互相认识的信息后，返回至少需要多少张桌子才能满足这个要求。分析该算法的时间复杂度。

(b) 现在考虑另一个场景：假设只有两张桌子，另外你知道有些客人不喜欢对方。如果 u 不喜欢 v，则必然 v 不喜欢 u。你的目标是将客人分配到两张桌子上，使得同一桌子上不存在两个客人不喜欢对方。例如，如果客人间不喜欢的关系如图(a)所示，则可以将 A,C 安排在一个桌子，B,D,E 安排在另一个桌子。如果客人间的不喜欢关系如图(b)所示，则这个目标无法达到。设计高效的算法，在给定客人间的不喜欢关系之后，返回目标是否可能达到。分析算法的时间复杂度。



(a) (b)

期末复习题

Zhilin Wu (吴志林),
State Key Laboratory of Computer Science,
Institute of Software,
Chinese Academy of Sciences

一、时间复杂度

(a) 将以下函数按照增长速度排序($\log n = \log_2 n$):

$$f_1(n) = n^{\pi} \qquad f_2(n) = \pi^n \qquad f_3(n) = \binom{n}{5} \qquad f_4(n) = \sqrt{2^{\sqrt{n}}}$$

$$f_5(n) = \binom{n}{n-4} \qquad f_6(n) = 2^{\log^4 n} \qquad f_7(n) = n^{5(\log n)^2} \qquad f_8(n) = n^4 \binom{n}{4}$$

$$f_1(n), f_5(n), f_3(n), f_8(n), f_7(n), f_6(n), f_4(n), f_2(n)$$

$$f_1(n) = n^{\pi} = o(n^4), f_5(n) = \Theta(n^4) \quad f_5(n) = \Theta(n^4), n^4 = o(n^5), f_3(n) = \Theta(n^5)$$

$$f_8(n) = \Theta(n^8), n^8 = o(n^{5\log^2 n}) \quad f_7(n) = o(n^{\log^3 n}), f_6(n) = 2^{\log^4 n} = n^{\log^3 n}$$

$$f_6(n) = 2^{\log^4 n}, f_4(n) = 2^{n/4}, \log^4 n = o(n/4)$$

$$f_4(n) = 2^{n/4} = o(2^n), 2^n = o(2^{(\log_2 \pi)n}), f_2(n) = \pi^n = 2^{(\log_2 \pi)n}$$

一、时间复杂度

(b) 求解以下分治算法分析产生的递推关系：

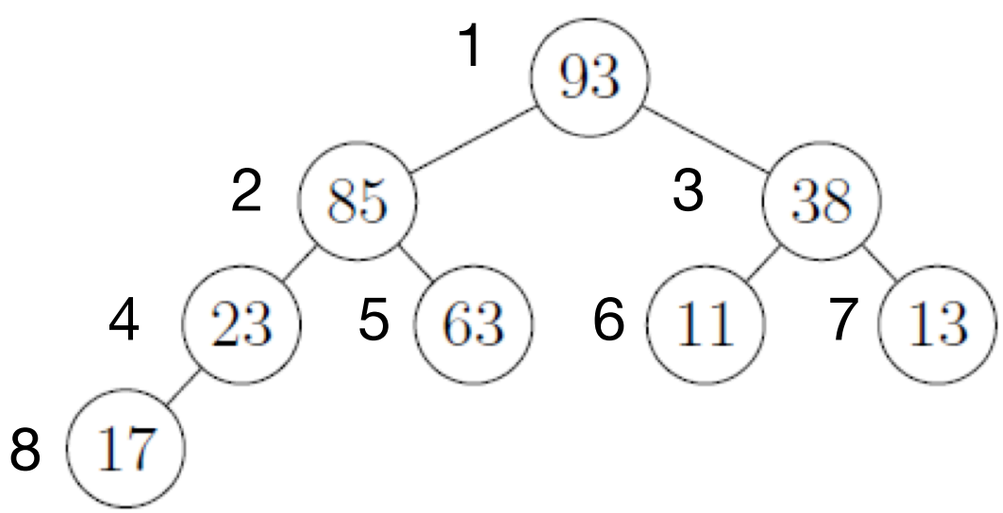
$$T(n) = T(\tfrac{n}{3}) + T(\tfrac{2n}{3}) + \Theta(n).$$

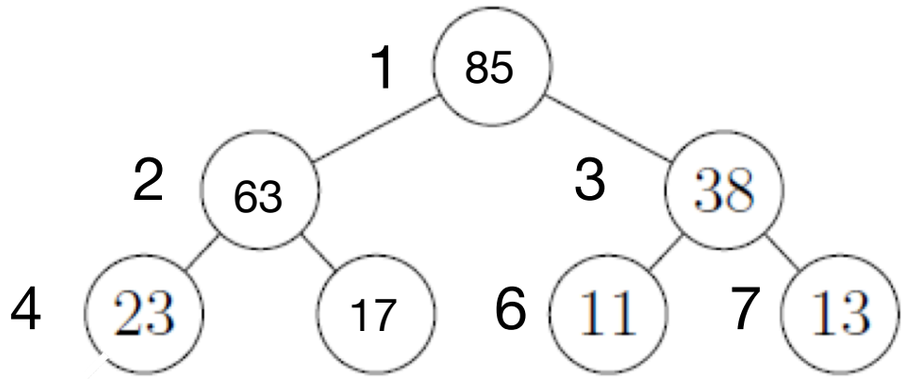
$$T(n) = T(n/3) + T(2n/3) + \Theta(n) =$$
$$T(n/3^2) + T(2n/3^2) + T(2n/3^2) + T(2^2n/3^2) + \Theta(n) + \Theta(n/3) + \Theta(2n/3)$$
$$= T(n/3^2) + T(2n/3^2) + T(2n/3^2) + T(2^2n/3^2) + 2\Theta(n)$$
$$\dots$$
$$= k\Theta(n) + \sum_{j=0}^{k} C_k^j T(n(1/3)^j(2/3)^{k-j}) =$$
$$\dots = O(n \log_{3/2} n)$$

二、堆排序算法

说明这个堆如何使用数组表达。如果这个堆里的最大元素被删除，剩下的堆的数组表达是什么？

删除93：首先调换93和17， 并删除93，然后MAX-HEAPIFY





93	85	38	23	63	11	13	17

85	63	38	23	17	11	13

期末复习题3-4

詹博华　2021/11/17

问题#3

- 我们称一个子序列$y_1, y_2, \dots, y_n$为交替序列，如果每个相邻的y_i, y_{i+1}, y_{i+2}都满足$y_i < y_{i+1} > y_{i+2}$或$y_i > y_{i+1} < y_{i+2}$。也就是说，如果$y_i < y_{i+1}$，则$y_{i+1} > y_{i+2}$，反之亦然。给定一个序列$x_1, x_2, \dots x_n$，我们想要找到这个序列的最长的交替子序列。
- 例如，如果初始序列是：

$$x_1 = 13, x_2 = 93, x_3 = 86, x_4 = 50, x_5 = 63, x_6 = 4,$$

这个序列的最长交替子序列的长度为5，由$13, 93, 50, 63, 4$组成。

- 使用动态规划设计寻找最长交替子序列的算法。

问题#3解答

- 我们定义子问题$DP(i,b)$，其中$1 \leq i \leq n$，b是一个布尔值。如果b为真，则$DP(i,b)$代表最长的终止于x_i的交替子序列，其中最后一步是上升的。如果b为假，$DP(i,b)$代表最长的终止于x_i的交替子序列，其中最后一步是下降的。如果子序列的长度为1，则我们定义它既是上升的也是下降的。例如$DP(5,\text{True}) = 4$，因为子序列x_1, x_2, x_4, x_5的长度为4，并且最后一步上升。$DP(5,\text{False}) = 3$，因为子序列x_1, x_2, x_5的长度为3，并且最后一步下降。

问题#3解答

a. 对于以上序列，计算$DP(i, b)$，对于每个$1 \leq i \leq 6$和$b \in \{\text{True}, \text{False}\}$，写入以下表中：

$$x_1 = 13, x_2 = 93, x_3 = 86, x_4 = 50, x_5 = 63, x_6 = 4$$

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
$b = \text{True}$	1	2	2	2	4	1
$b = \text{False}$	1	1	3	3	3	5

问题#3解答

b. 写出$DP(i,b)$的递归关系。

$$DP(i,\textsc{True}) = 1 + \max_{1\le j<i \text{ and } x_i>x_j} DP(j,\textsc{False})$$

$$DP(i,\textsc{False}) = 1 + \max_{1\le j<i \text{ and } x_i<x_j} DP(j,\textsc{True})$$

c. 写出$DP(i,b)$递归关系的基本情况（$i=1$）。

$$DP(i,\textsc{True}) = 1 \qquad \text{if } x_i = \min\{x_1,\ldots,x_i\}$$

$$DP(i,\textsc{False}) = 1 \qquad \text{if } x_i = \max\{x_1,\ldots,x_i\}$$

问题#3解答

d. 如果采用自底向上的计算方式，给出一个合适的计算顺序。
   按照i从小到大的顺序，计算每个$DP(i, \text{True})$和$DP(i, \text{False})$。
e. 如果计算了所有的$DP(i, b)$，如何计算最长交替子序列的长度？
   需要取表中所有值的最大值。

问题#3解答

f. 将以上步骤放到一起，给出计算最长交替子序列长度的算法，使用伪代码表示。分析算法的时间复杂度。

for $i = 1$ **to** n
  for b = {True, False}
    if $DP(i, b)$ in base case:
      compute $DP(i, b)$ using c)
    else
      compute $DP(i, b)$ using b)
return maximum value in $DP(i, b)$

复杂度分析：
由于b)和c)中的计算需要线性时间，整个计算需要$O(n^2)$时间。

问题#4

- 给定一个不带括号的，由加法和乘法组成的表达式，找出如何添加括号，使得表达式的取值最大。
- 例如，如果提供的表达式是$6 + 0 \cdot 6$，则应该添加括号为$(6 + 0) \cdot 6$，结果为36，而不是$6 + (0 \cdot 6)$，结果为6。再比如，如果提供的表达式是$0.1 \cdot 0.1 + 0.1$，则应该添加括号为$(0.1 \cdot 0.1) + 0.1$，结果为0.11，而不是$0.1 \cdot (0.1 + 0.1)$，结果为0.02。
- 使用动态规划设计多项式时间的算法，在给定表达式之后，找到最好的添加括号的方法。假设输入的格式为$x_0, o_0, x_1, o_1, \ldots o_{n-1}, x_n$，其中每个$x_i$是一个数字，每个$o_i$是加号或乘号。分析算法的时间复杂度。

问题#4解答

- 设$DP[i,j]$为加括号后$x_i, o_i, \ldots x_j$的可能最大值。
- 递归关系为:

$$DP[i,j] = \max_{k=i}^{j-1} \Big(DP[i,k] \; o_k \; DP[k+1,j] \Big).$$

- 基本情况为:

$$DP[i,i] = x_i.$$

- 计算顺序可以按照$j - i$从小到大的方式。

问题#4解答

- 伪代码

for $i = 0$ **to** n
  $DP[i, i] = x_i$
for $l = 0$ **to** n
  for $i = 0$ **to** $n - l$
    $j = i + l$
    compute $DP[i, j]$ using recurrence
return $DP[0, n]$

- **时间复杂度：**每个子问题的计算$O(n)$，一共$O(n^2)$个子问题，因此一共$O(n^3)$。

问题#4解答

- **出题时忽略的一个细节：**以上答案假设所有涉及的数字都是正数，因此最大化$a+b$和$a\times b$只需要最大化a和b。但如果a和b可能是负数，以上假设不成立。
- **解决方案：**除了维护$DP_{\max}[i,j]$为$x_i, o_i, \dots x_j$加括号后的最大值以外，还维护$DP_{\min}[i,j]$为$x_i, o_i, \dots x_j$加括号后的最小值。在递归关系中，考虑$DP[i,k]$和$DP[k+1,j]$分别为最大值和最小值的四种情况。算法的时间复杂度依然是$O(n^3)$。

Algorithm Design and Analysis

David N. Jansen

名 姓

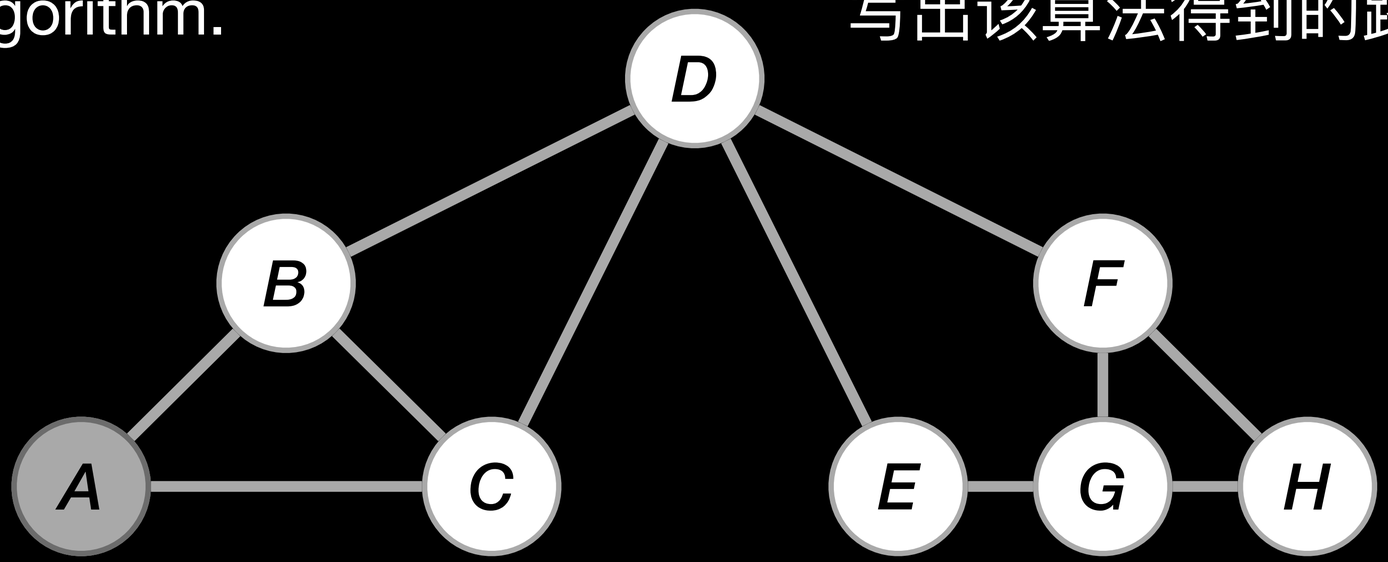
算法设计与分析

杨大卫

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。

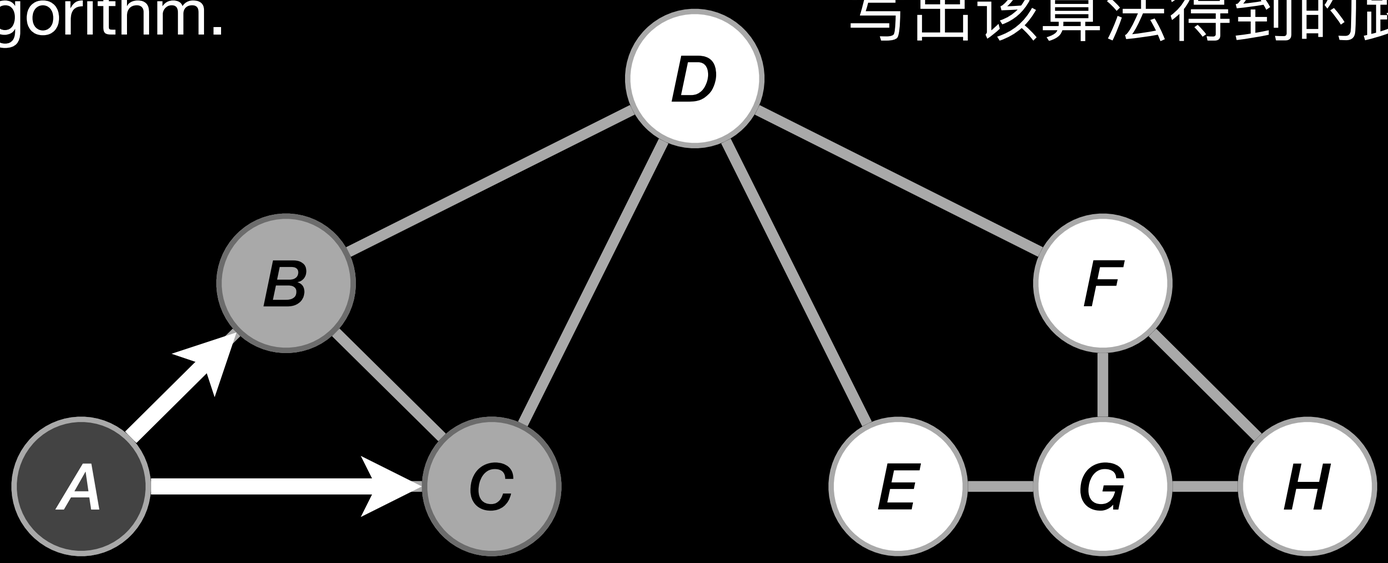


Queue
A

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。

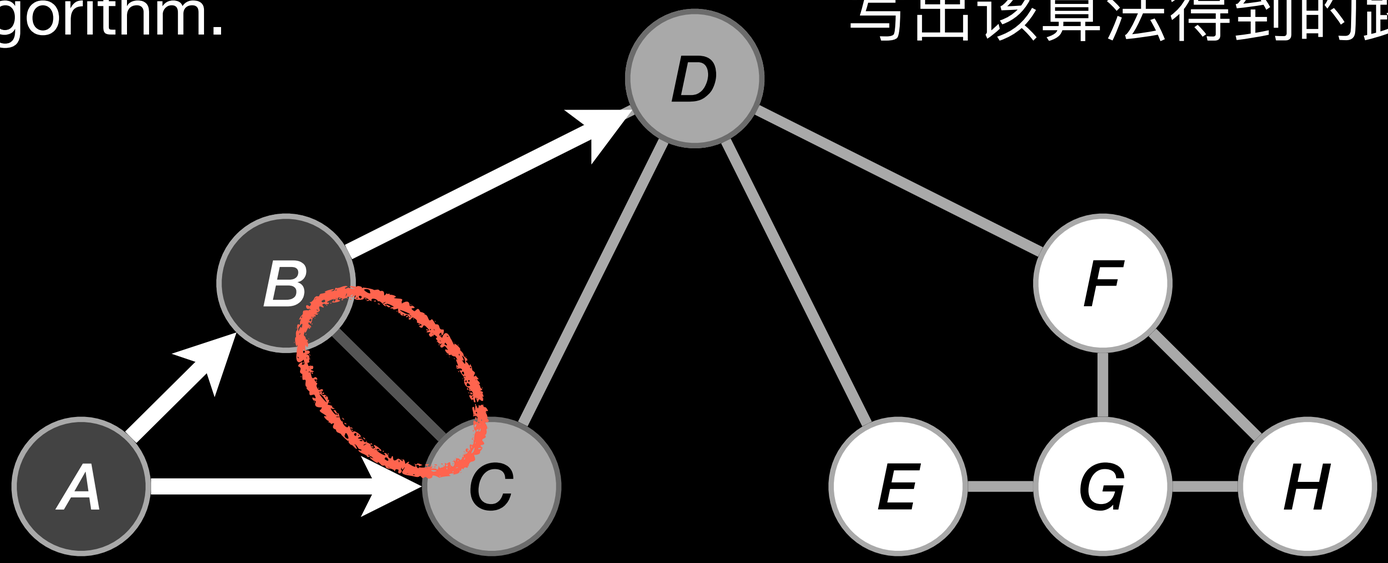


Queue
B
C

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。

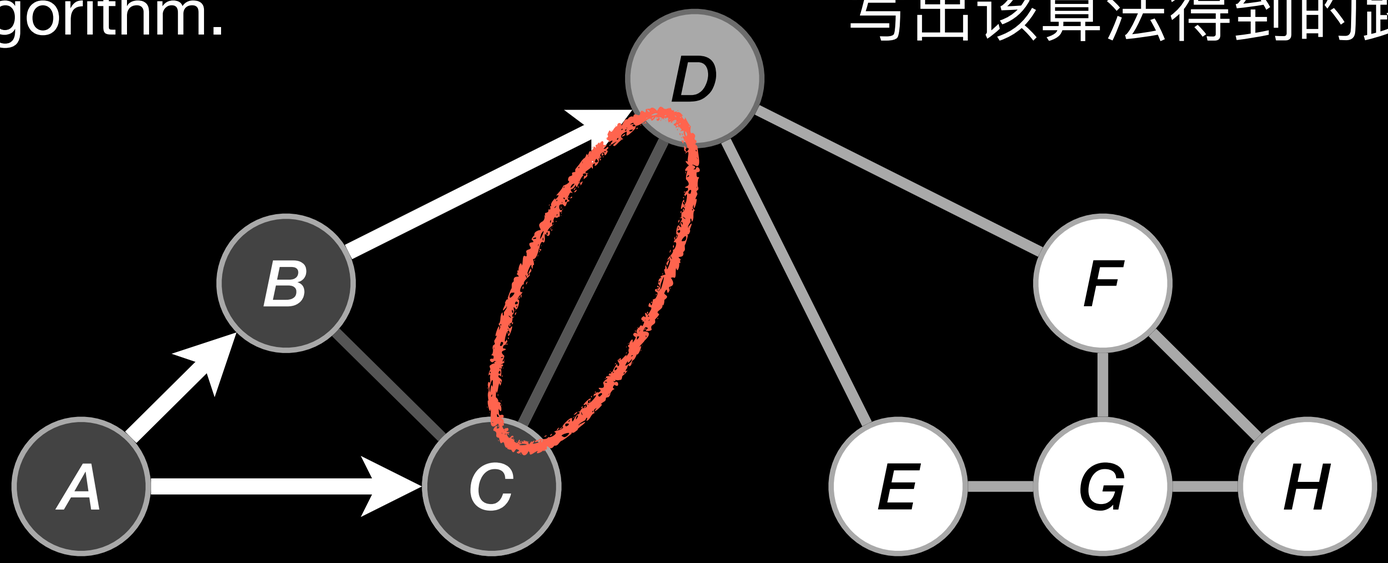


Queue
C
D

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。

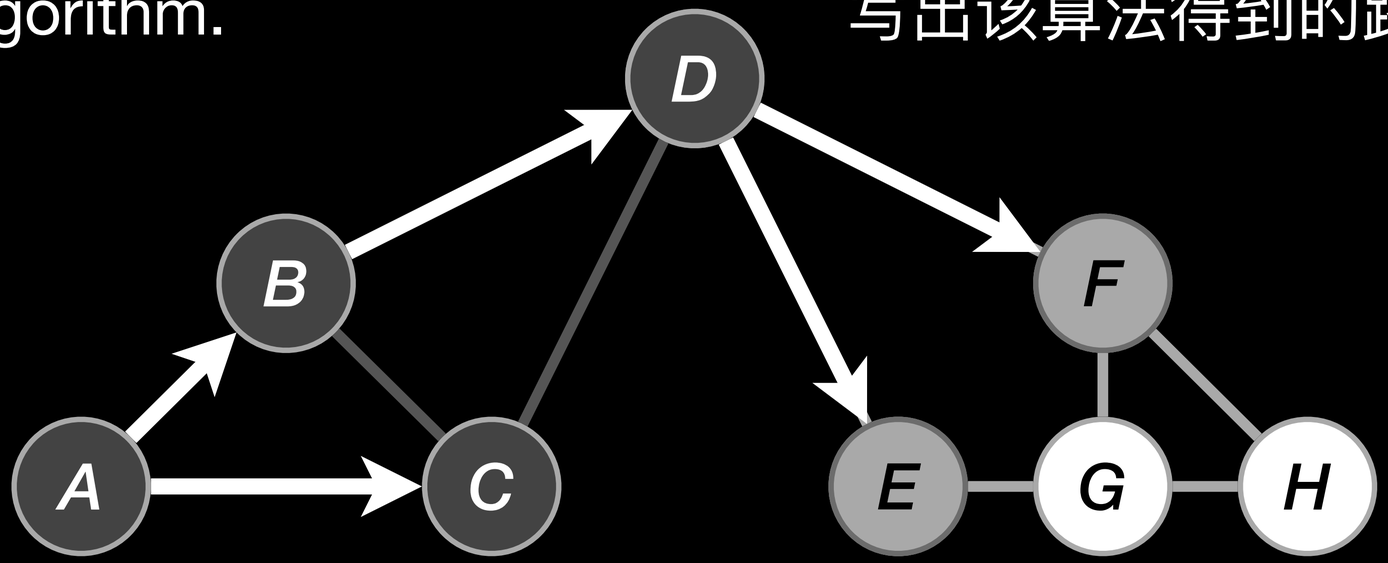


Queue
D

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。

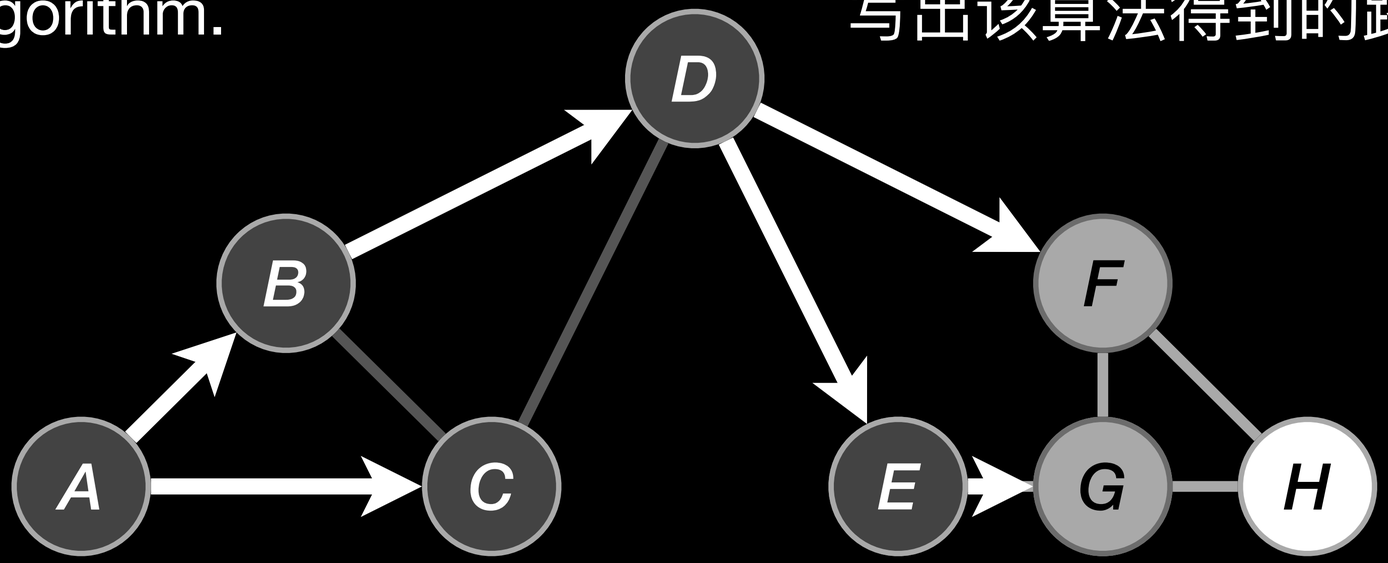


Queue
E
F

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。

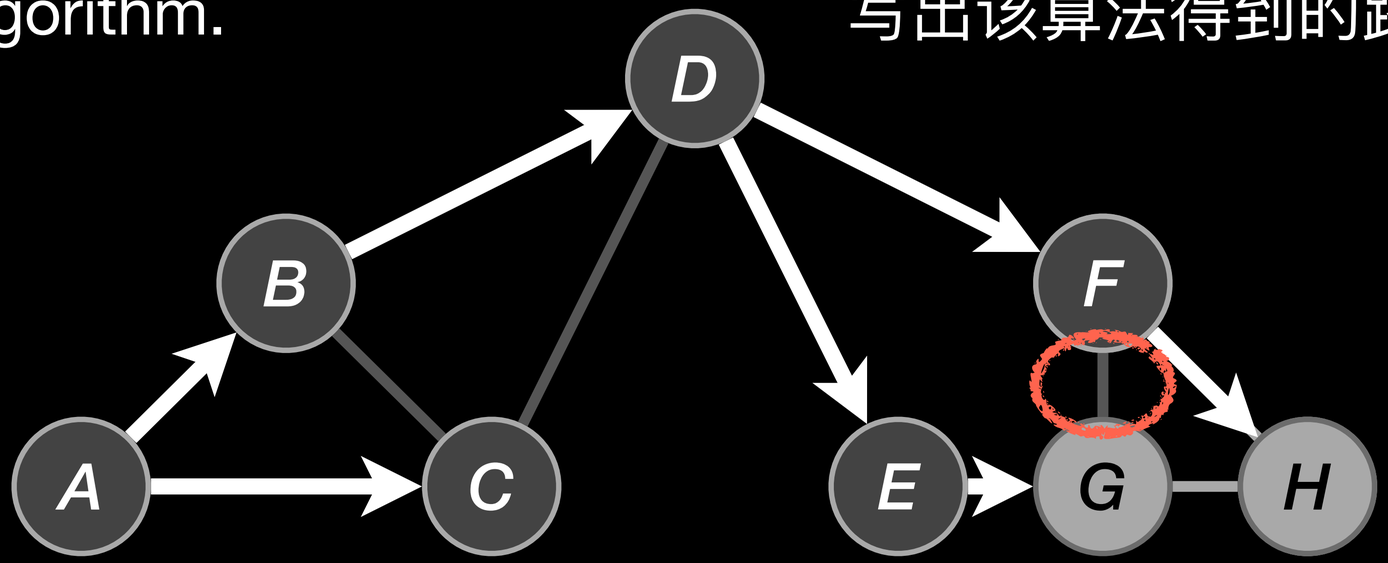


Queue
F
G

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。

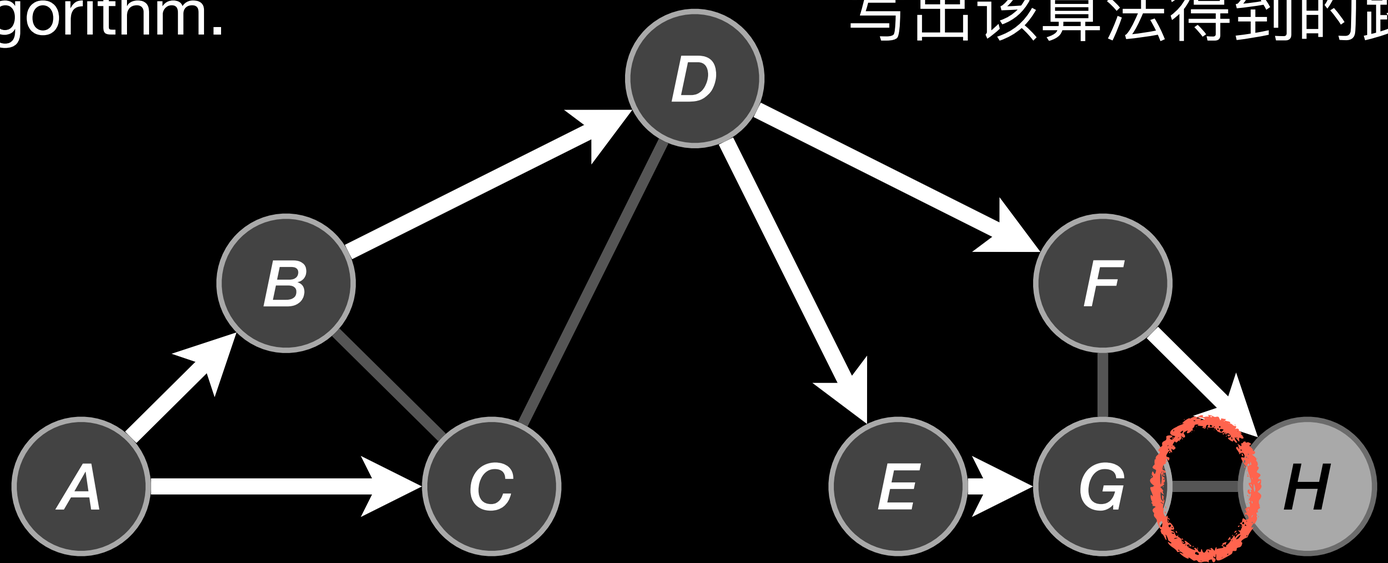


Queue
G
H

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。

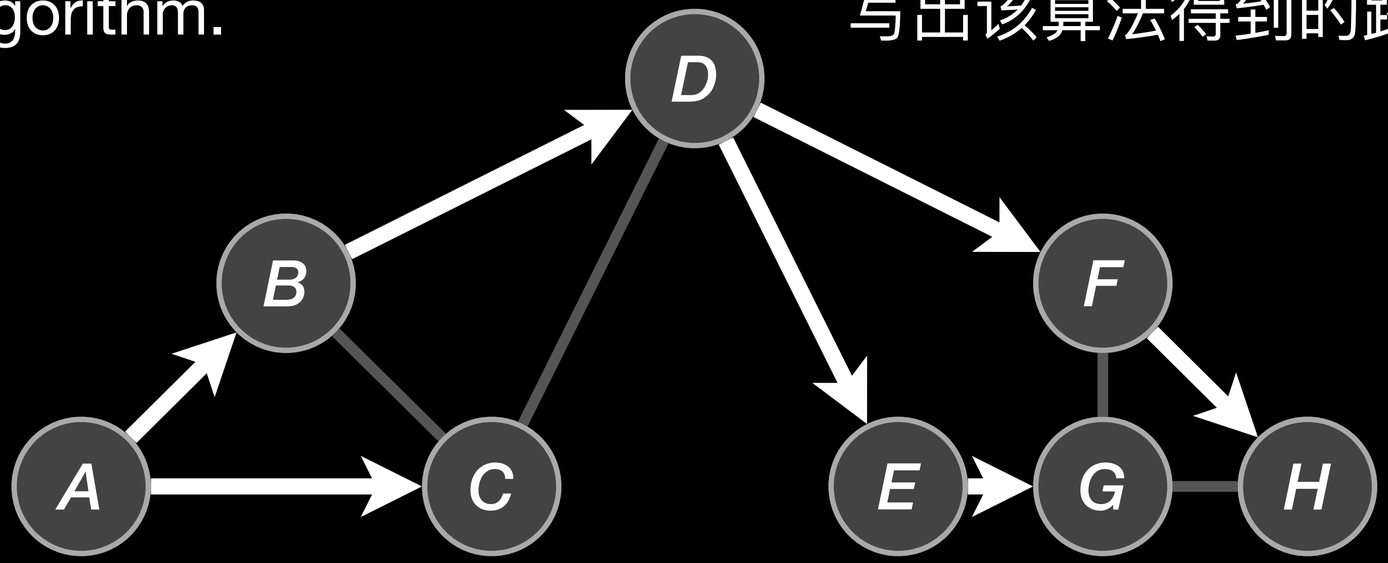


Queue
H

5. Graph Traversal　五、图的遍历

(a) We use breadth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(a) 假设我们使用广度优先搜索寻找从*A*到*H*的路径，写出该算法得到的路径。



Queue

Path found: *A* → *B* → *D* → *F* → *H*

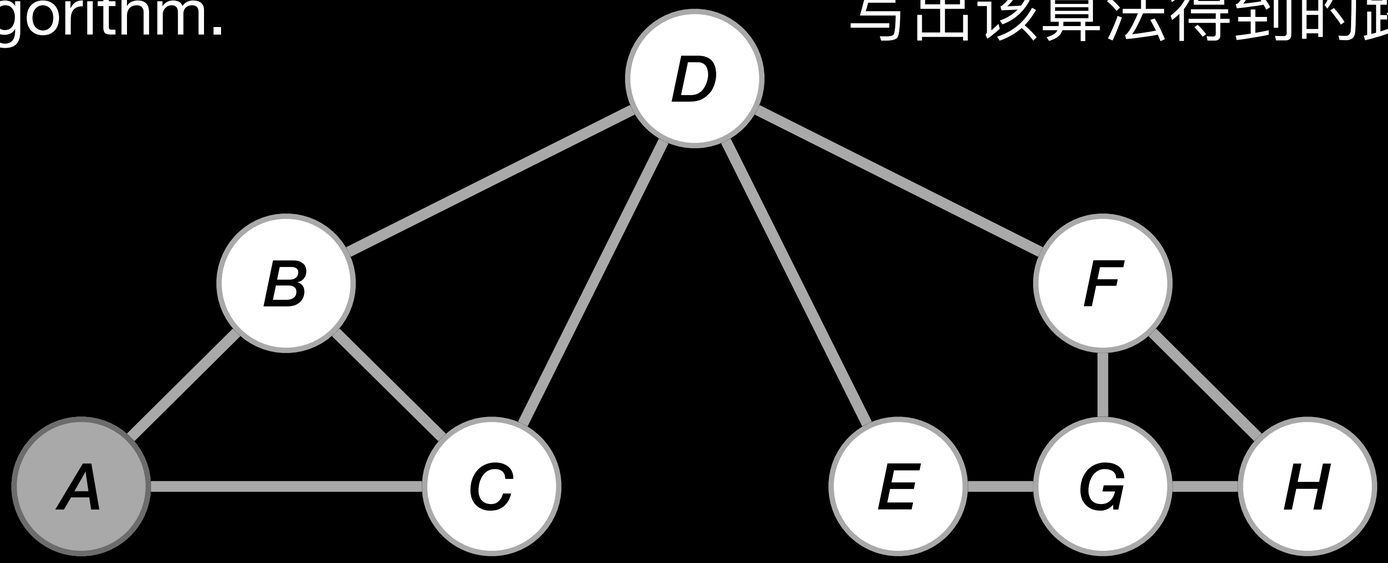
得到的路径：*A* → *B* → *D* → *F* → *H*

5. Graph Traversal　　五、 图的遍历

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

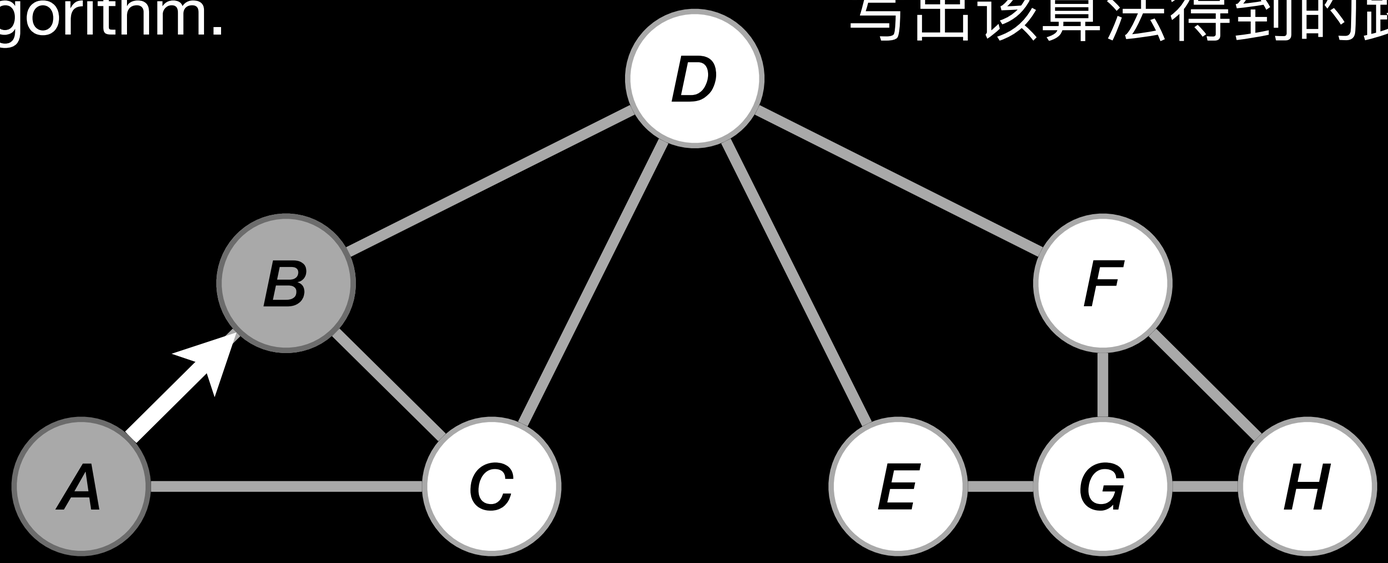


DFS(图) → DFS-VISIT(图,*A*)

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

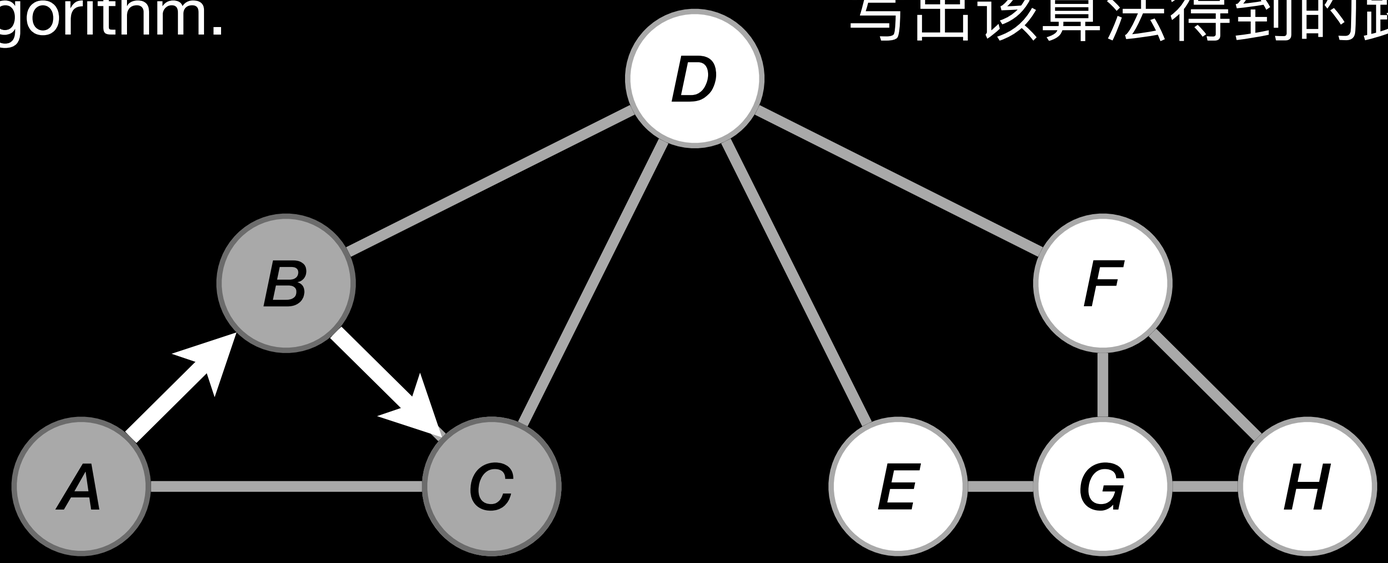


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*)

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

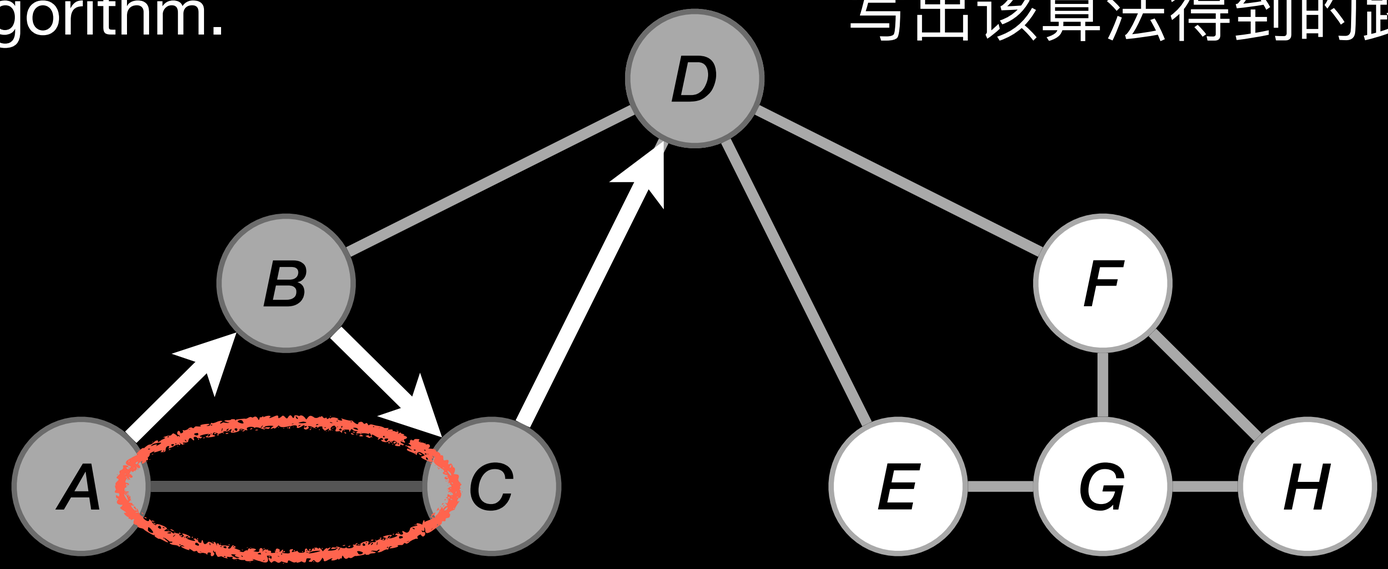


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*)

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

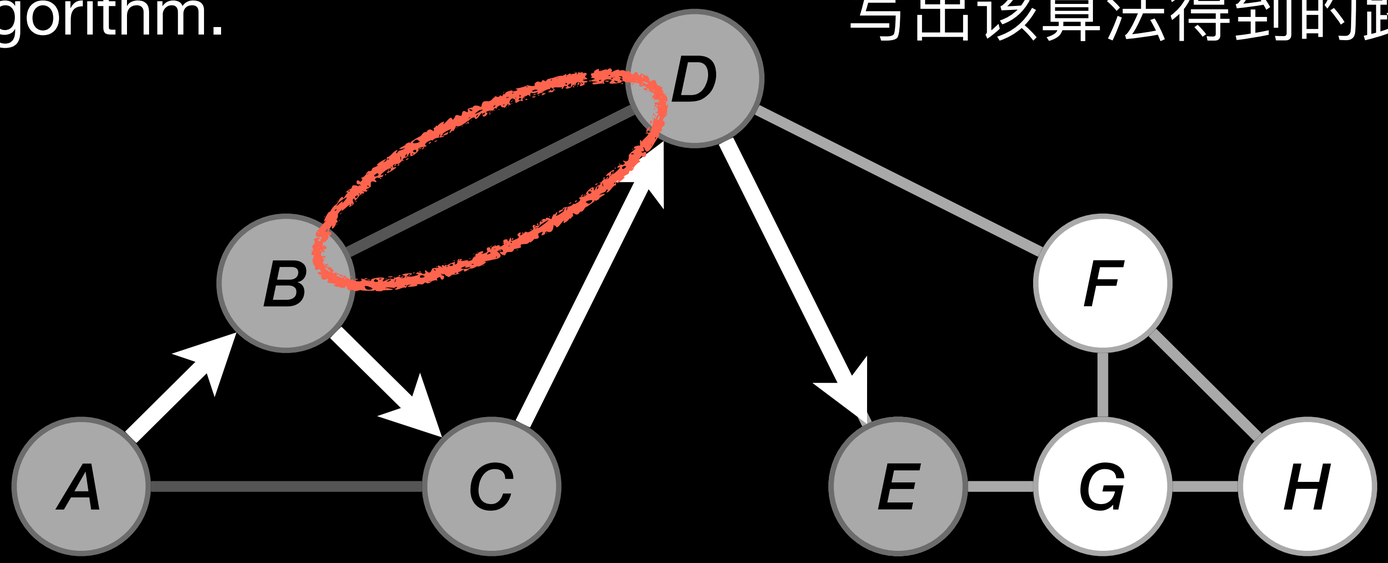


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

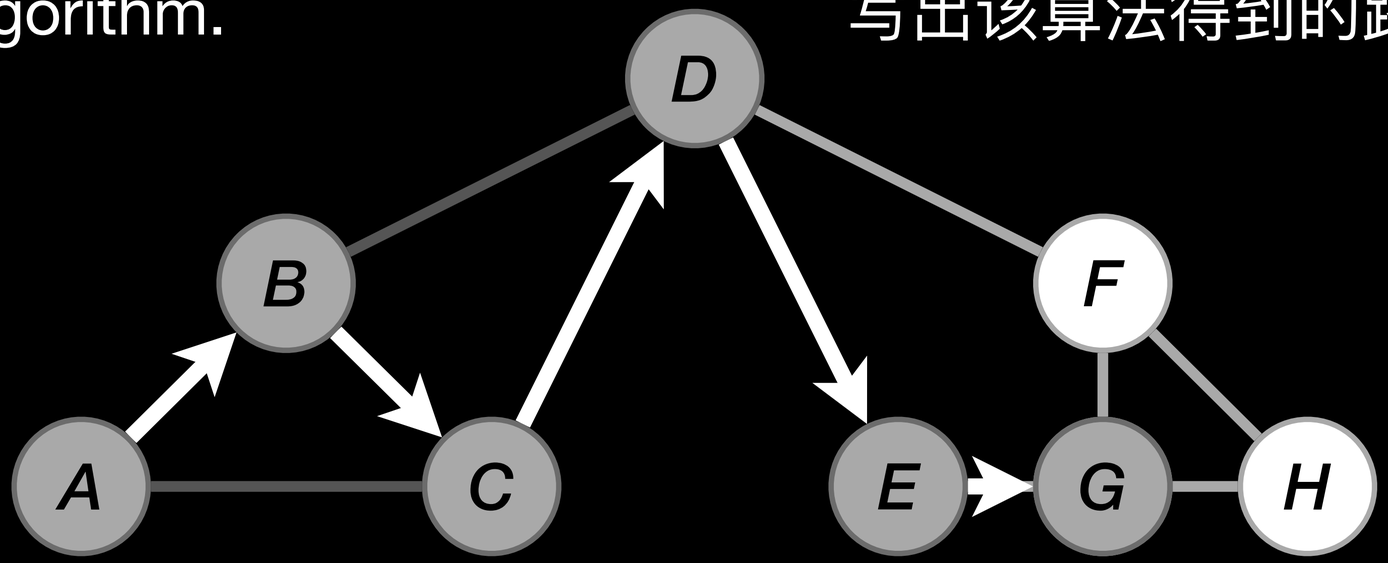


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)
→ DFS-VISIT(图,*E*)

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

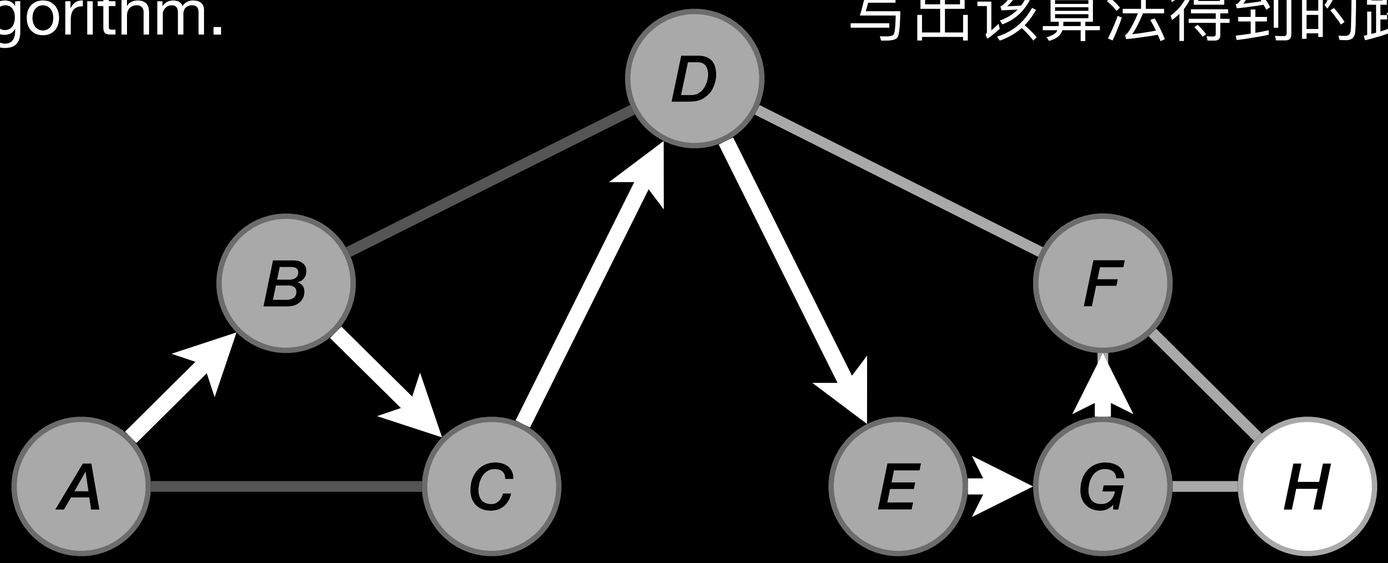


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)
→ DFS-VISIT(图,*E*) → DFS-VISIT(图,*G*)

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

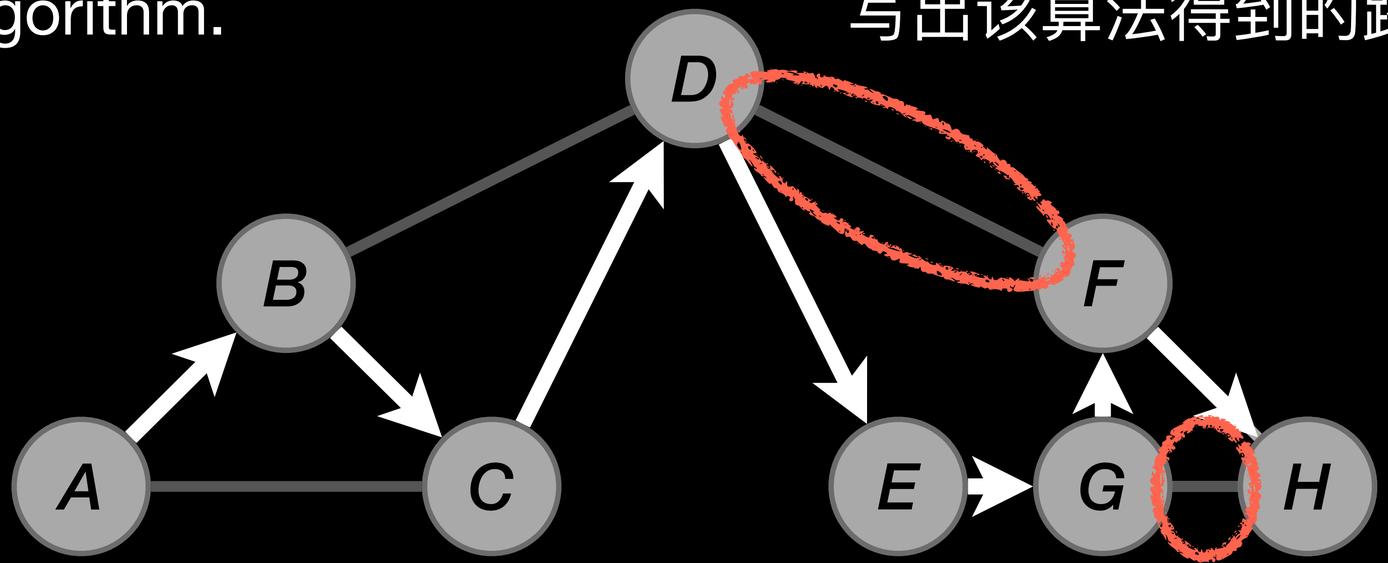


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)
→ DFS-VISIT(图,*E*) → DFS-VISIT(图,*G*) → DFS-VISIT(图,*F*)

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

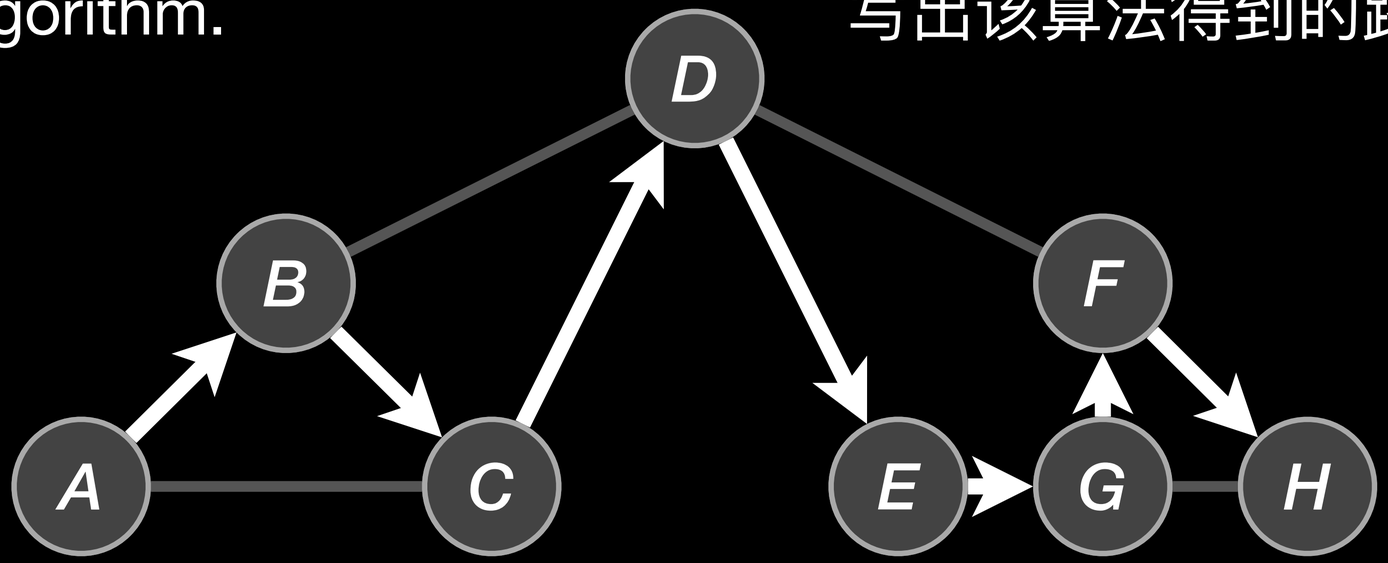


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)
→ DFS-VISIT(图,*E*) → DFS-VISIT(图,*G*) → DFS-VISIT(图,*F*) → DFS-VISIT(图,*H*)

5. Graph Traversal　　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。

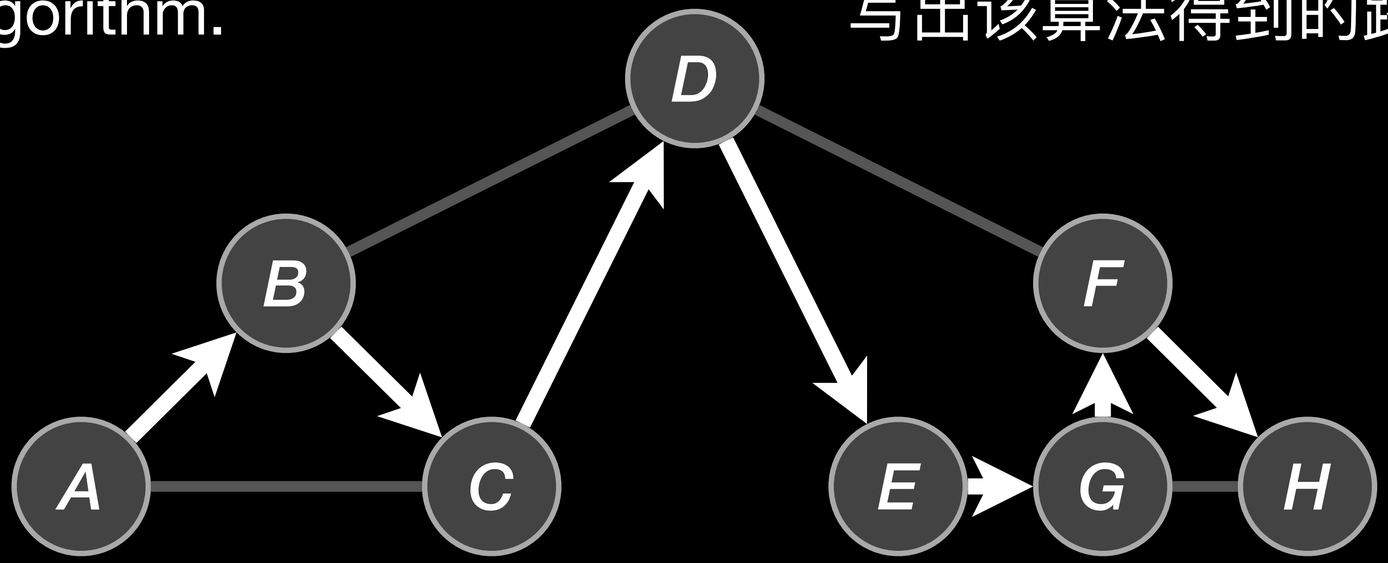


DFS(图)

5. Graph Traversal　五、图的遍历

(b) We use depth-first search to find a path from *A* to *H*. Write down the path found by the algorithm.

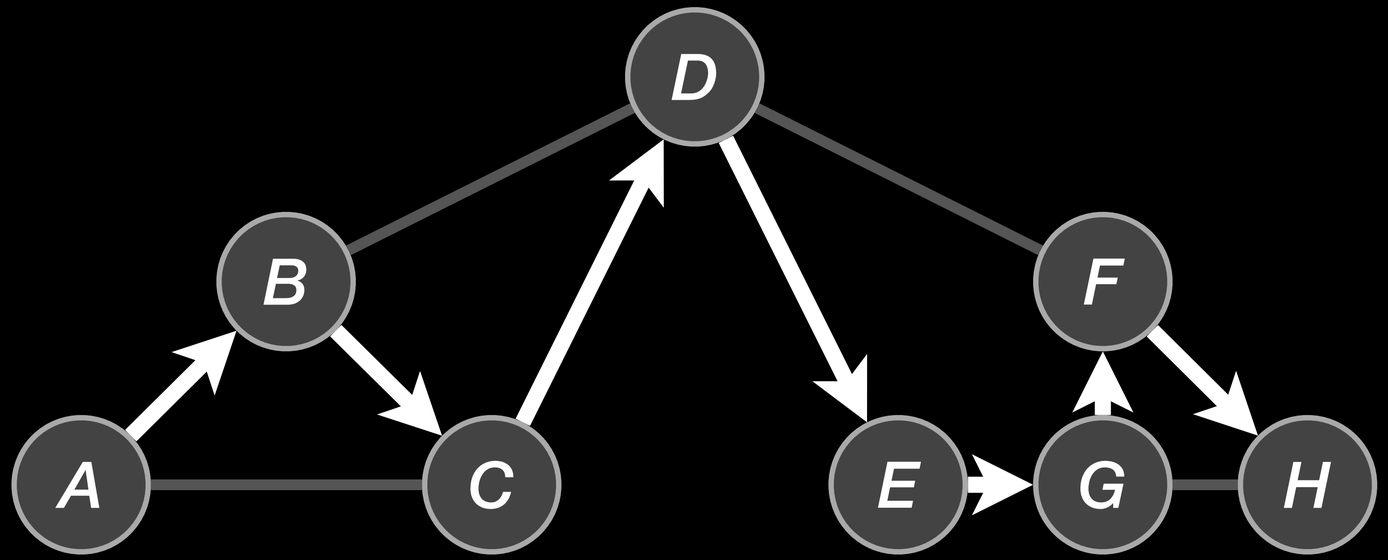
(b) 假设我们使用深度优先搜索寻找从到的路径，
写出该算法得到的路径。



Path found: *A*→*B*→*C*→*D*→*E*→*G*→*F*→*H*

得到的路径：*A*→*B*→*C*→*D*→*E*→*G*→*F*→*H*

5. Graph Traversal　　五、图的遍历

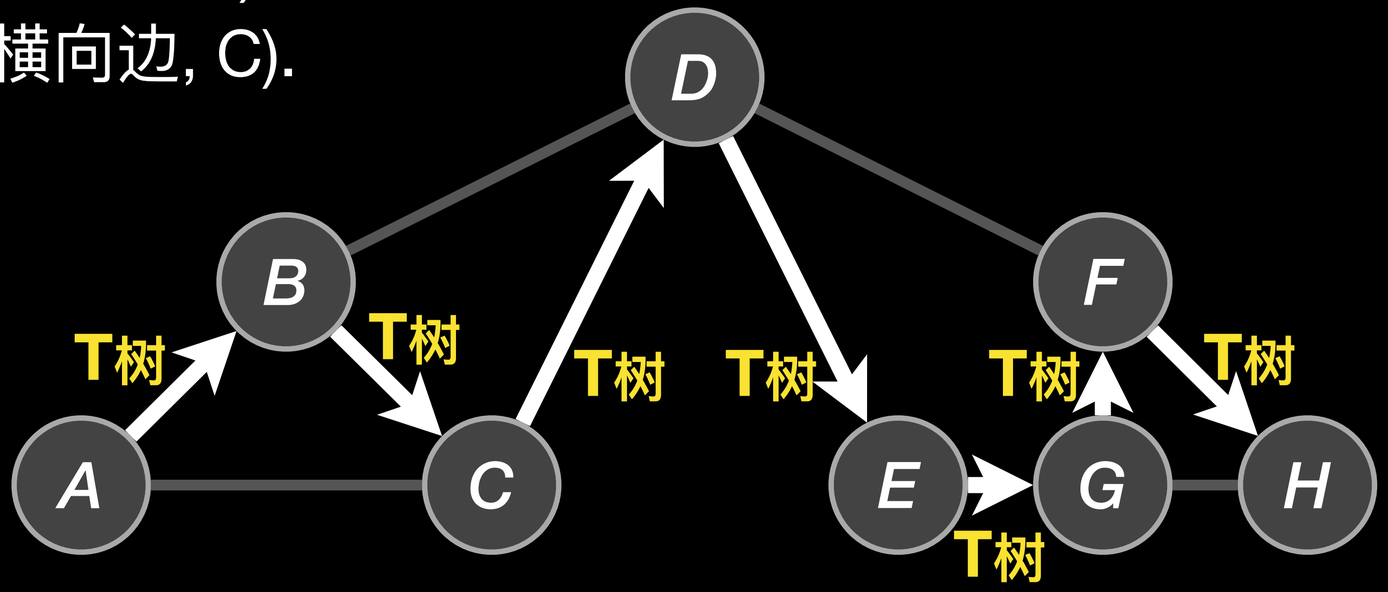


5. Graph Traversal　五、图的遍历

(c) Assume that we start a depth-first search from *A*. Mark each edge as tree edge (树边, T), back edge (后向边, B), forward edge (前向边, F), or cross edge (横向边, C).

(c) 假设我们从*A*开始采用深度优先搜索，将每条边标记为树边 (tree edge, T)、后向边 (back edge, B)、前向边 (forward edge, F)、和横向边 (cross edge, C)。

Tree edges are those followed by DFS.

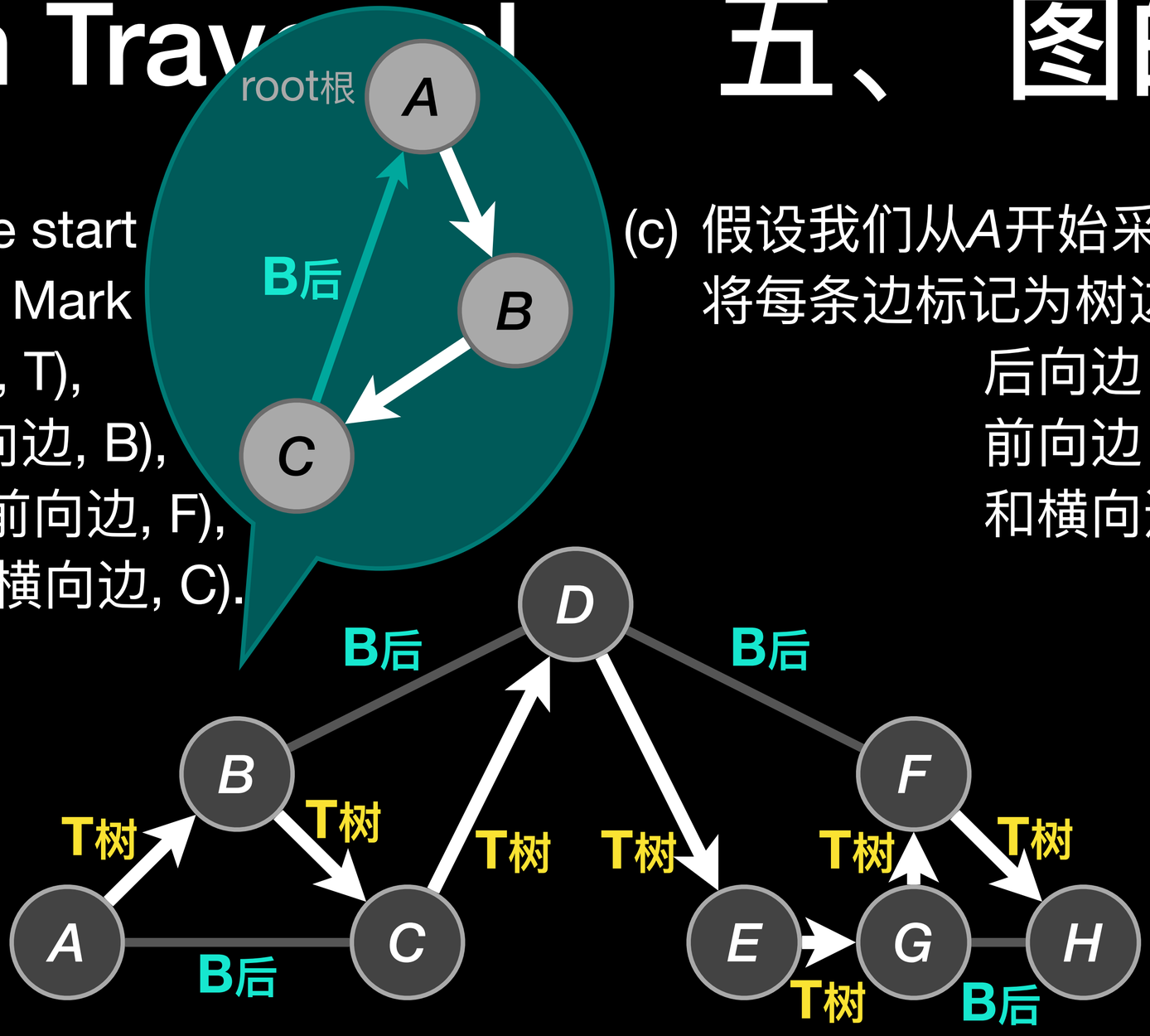


树边是所DFS遵循的边。

5. Graph Traversal 五、图的遍历

(c) Assume that we start search from *A*. Mark tree edge (树边, T), back edge (后向边, B), forward edge (前向边, F), or cross edge (横向边, C).

(c) 假设我们从*A*开始采用深度优先搜索，将每条边标记为树边 (tree edge, T)、后向边 (back edge, B)、前向边 (forward edge, F)、和横向边 (cross edge, C)。



DFS tries other edges in the direction from child back to parent.

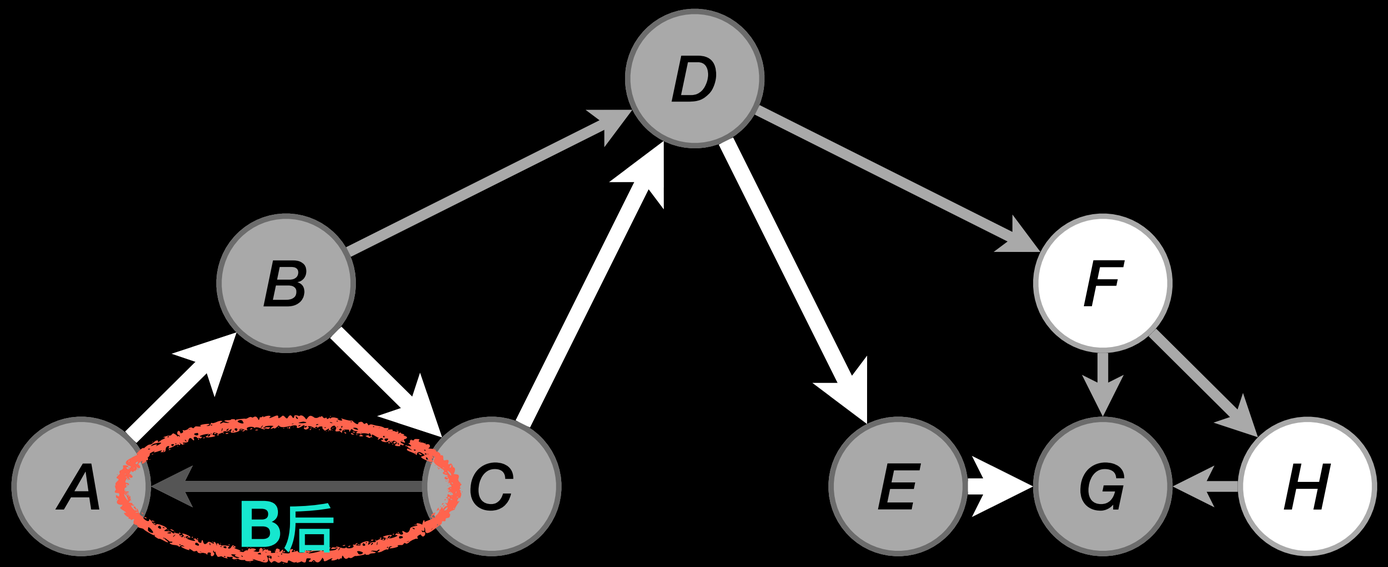
DFS尝试其他边使用从子到父的方向。

5. Graph Traversal　　五、图的遍历

5. Graph Traversal　五、图的遍历

(d) Do the same as in problem (c), but use the directed graph below:

(d) 假重复问题(c)，但使用以下的有向图：

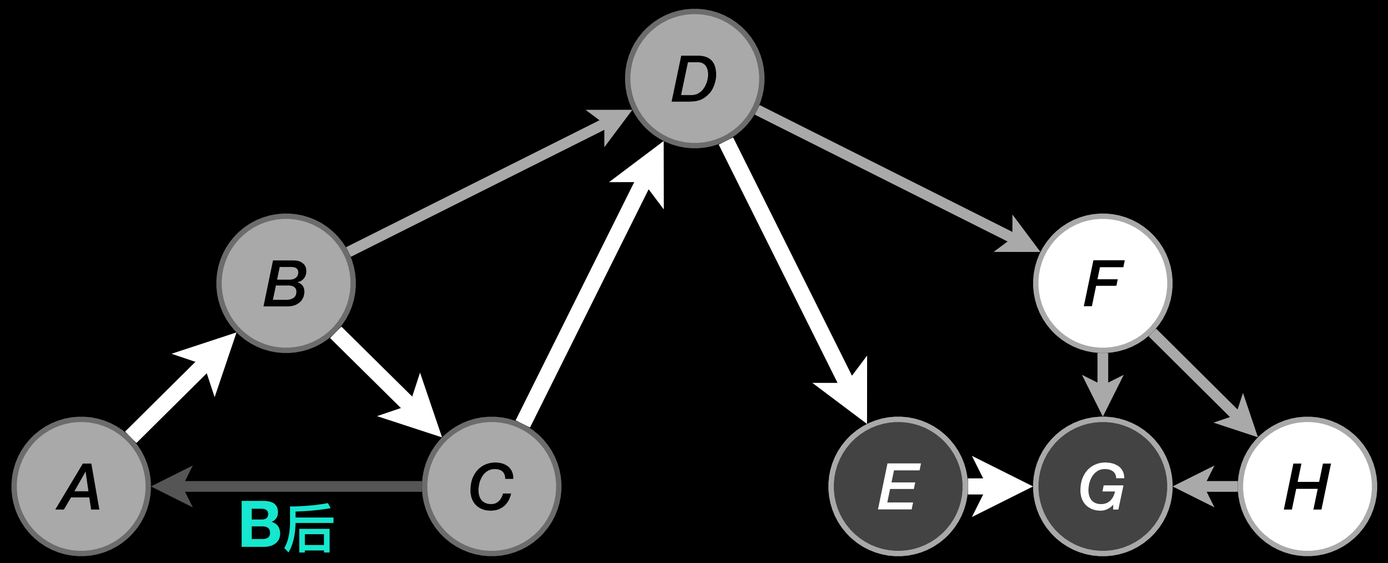


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)
→ DFS-VISIT(图,*E*) → DFS-VISIT(图,*G*) ↛ DFS-VISIT(图,*F*)

5. Graph Traversal　五、图的遍历

(d) Do the same as in problem (c), but use the directed graph below:

(d) 假重复问题(c)，但使用以下的有向图：

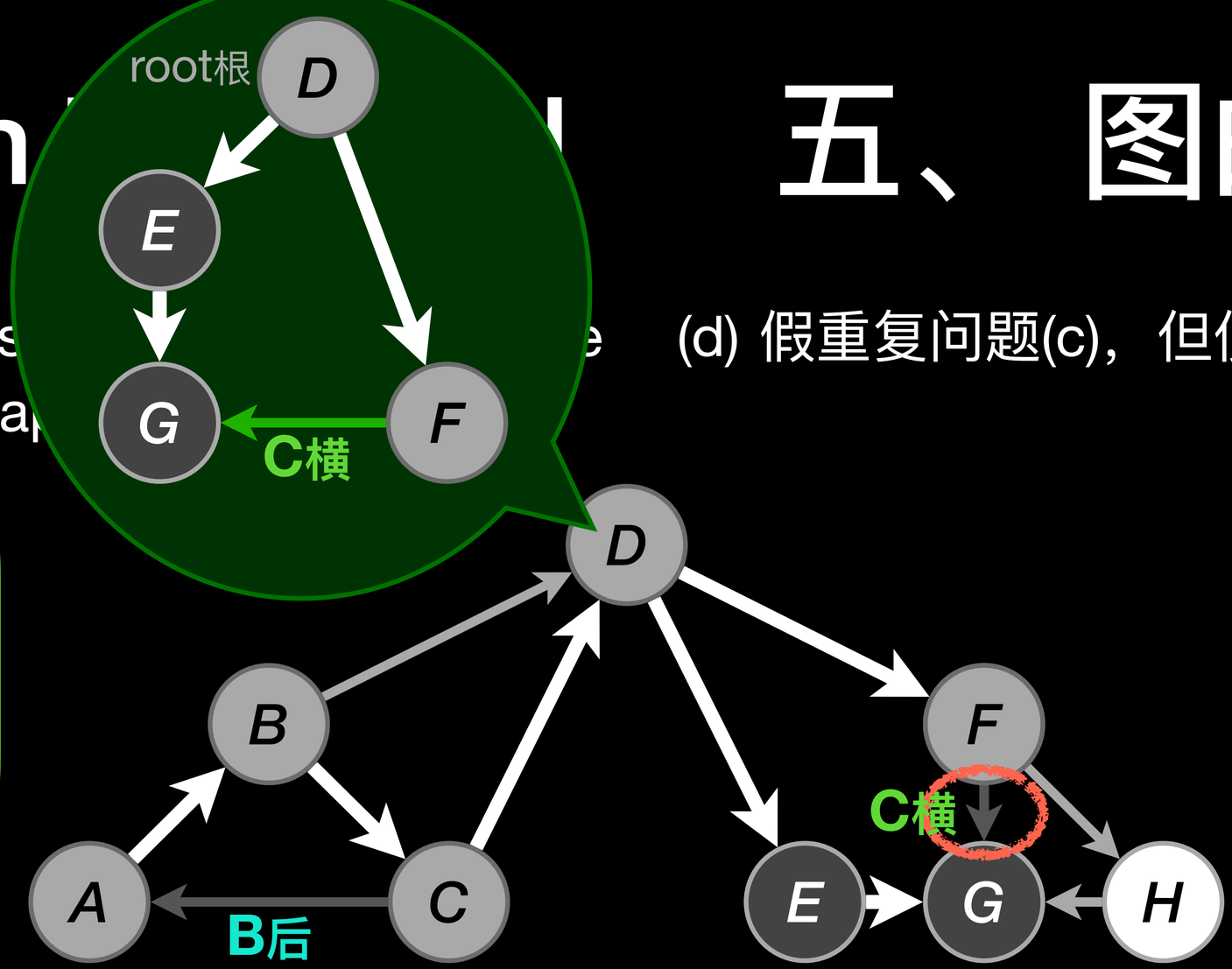


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)

5. Graph [illegible] 五、图的遍历

(d) Do the same as [illegible] the directed gra[illegible]

(d) 假重复问题(c)，但使用以下的有向图：



Cross edges can be drawn across/ horizontally.

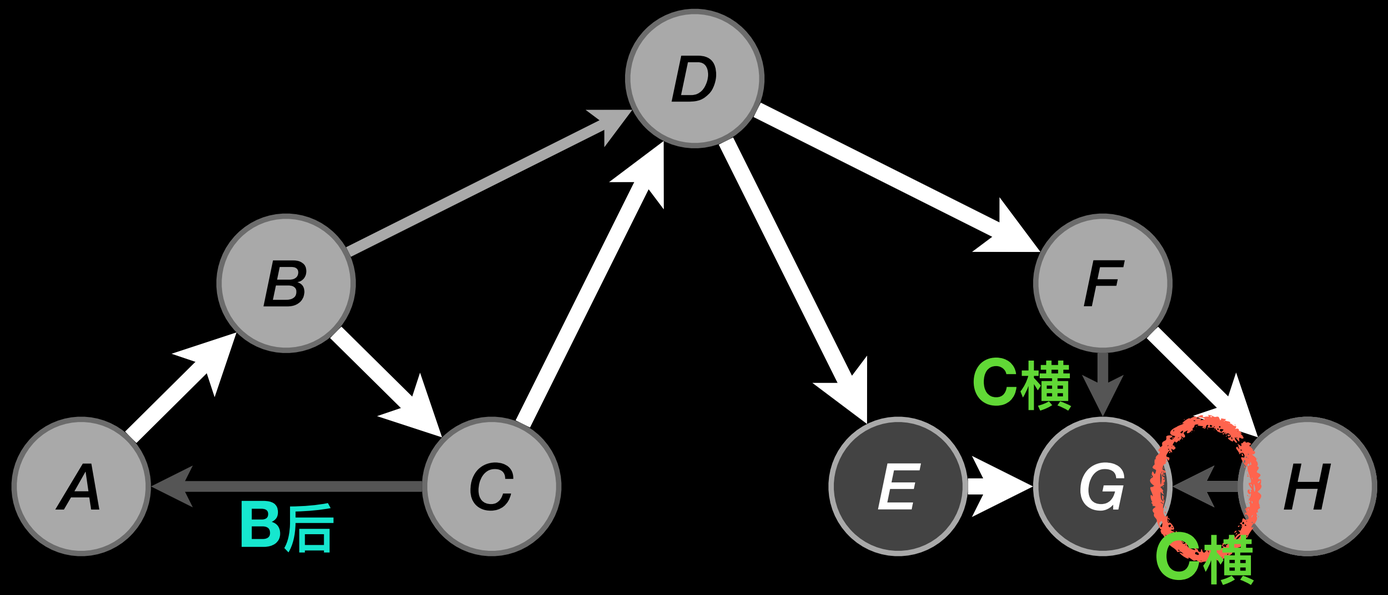
横向边可以横地绘制。

DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)
→ DFS-VISIT(图,*F*)

5. Graph Traversal　五、图的遍历

(d) Do the same as in problem (c), but use the directed graph below:

(d) 假重复问题(c)，但使用以下的有向图：

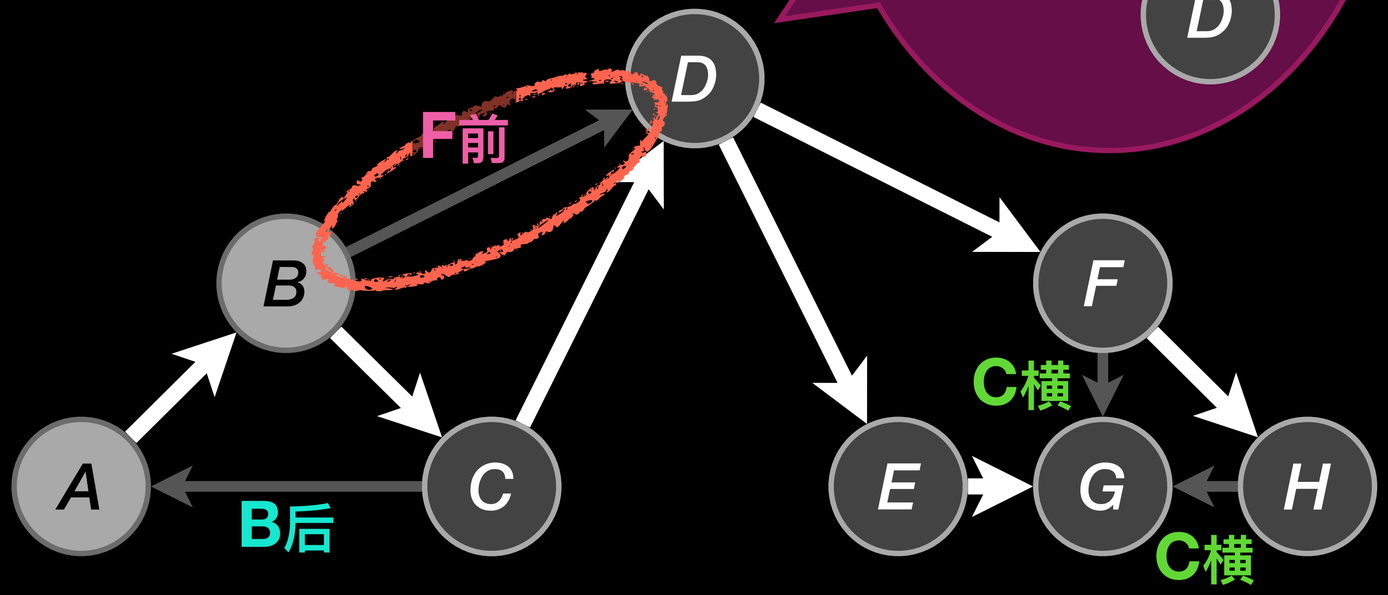


DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*) → DFS-VISIT(图,*C*) → DFS-VISIT(图,*D*)
→ DFS-VISIT(图,*F*) → DFS-VISIT(图,*H*)

5. Graph Traversal 图的遍历

(d) Do the same as in problem (c), but use the directed graph below:

(d) 使用以下的有向图：



Forward edges go from grandparent to grandchild.

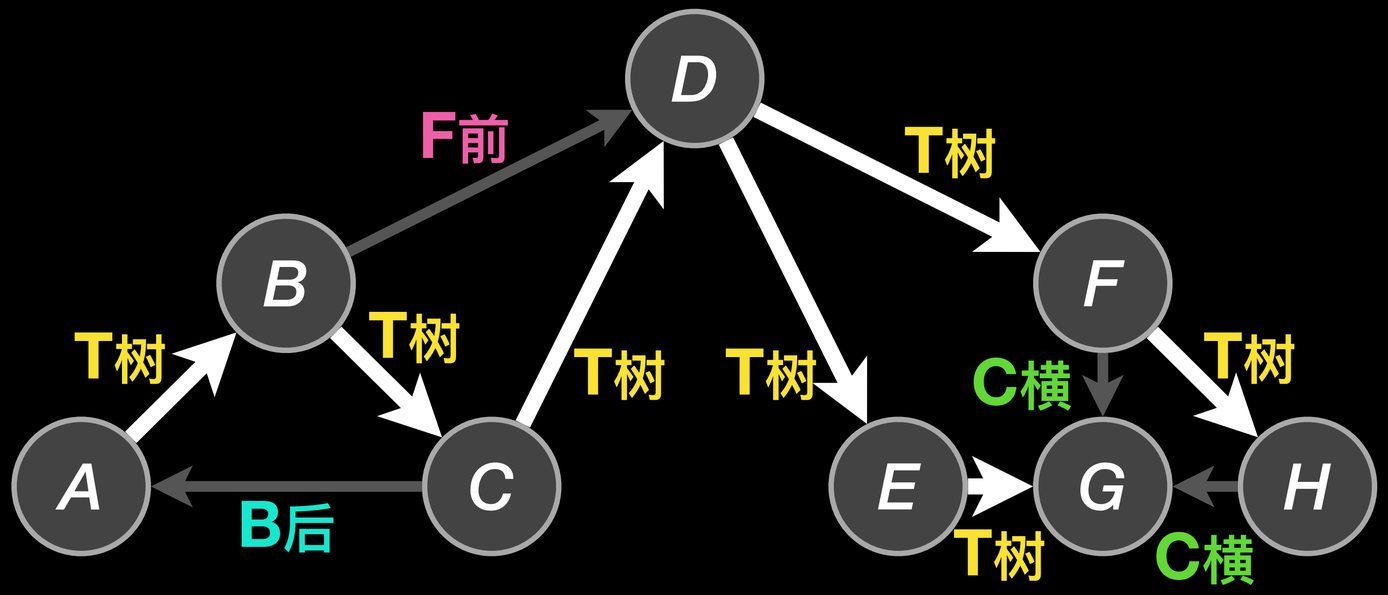
前向边从祖父母到孙子。

DFS(图) → DFS-VISIT(图,*A*) → DFS-VISIT(图,*B*)

5. Graph Traversal　五、图的遍历

(d) Do the same as in problem (c), but use the directed graph below:

(d) 假重复问题(c)，但使用以下的有向图：

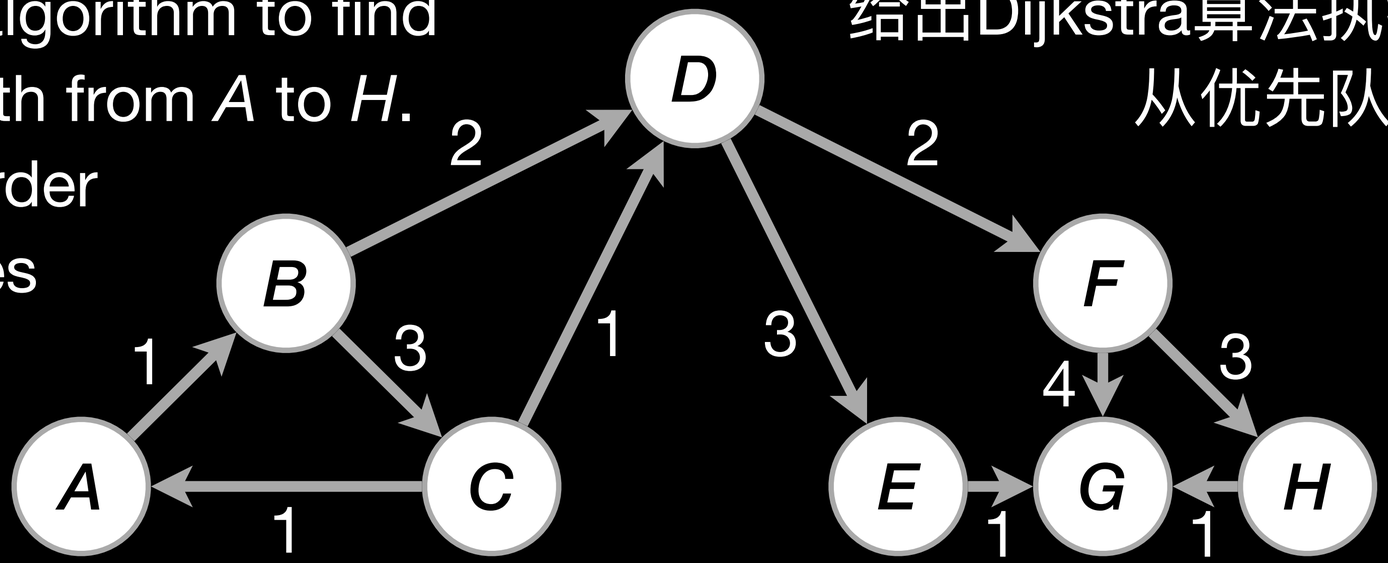


5. Graph Traversal　　五、图的遍历

5. Graph Traversal　五、图的遍历

(e) Now suppose that each edge of the graph has the following distance: Use Dijkstra's algorithm to find the shortest path from *A* to *H*. Also give the order in which vertices are removed from the priority queue.

(e) 现在假设图的每条边有一个距离如下：使用Dijkstra算法寻找到的最短路径，给出Dijkstra算法执行中顶点从优先队列中移出的顺序。



5. Graph Traversal

五、 图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

(e) 使用Dijkstra算法寻找到的最短路径。

Main Ideas of Dijkstra's Algorithm

- Start a shortest-path tree from *A*.
- Always add the vertex closest to *A* to the tree.
- Use a priority queue to find the closest vertex.
- Priority of *X* = known distance from *A* to *X*
- Known distance only uses edges that start in a tree vertex.

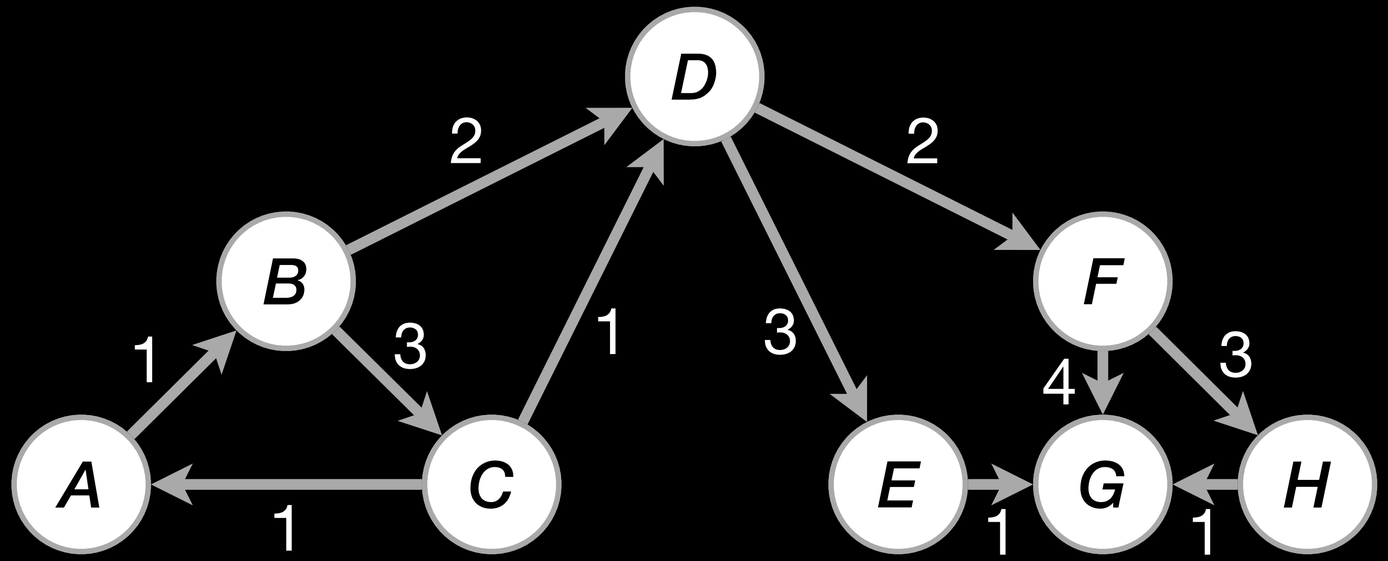
Dijkstra算法的主要思想

- 从*A*开始创建最短路径树。
- 始终将最接近*A*的顶点插入到树中。
- 使用优先队列查找最近的顶点。
- *X*点的优先级 = 从*A*到*X*的已知距离
- 已知距离仅使用从树顶点开始的边。

5. Graph Traversal　五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

(e) 使用Dijkstra算法寻找到的最短路径。



0

A

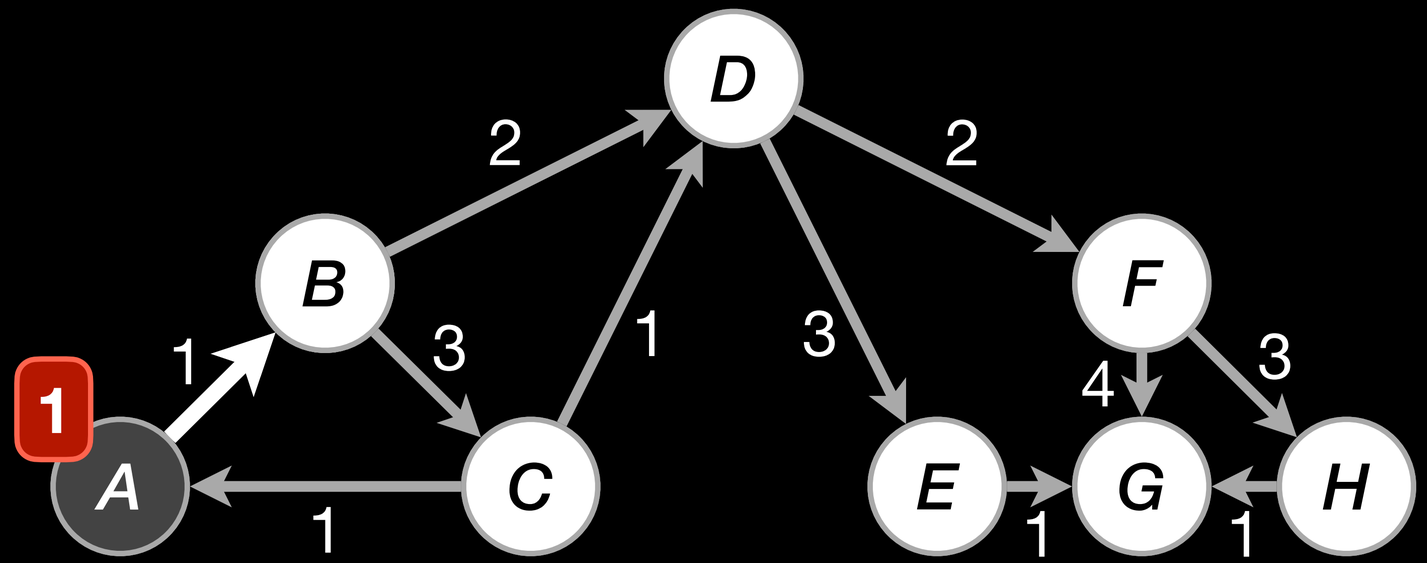
∞

B C D E F G H

5. Graph Traversal　五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

(e) 使用Dijkstra算法寻找到的最短路径。



1

B

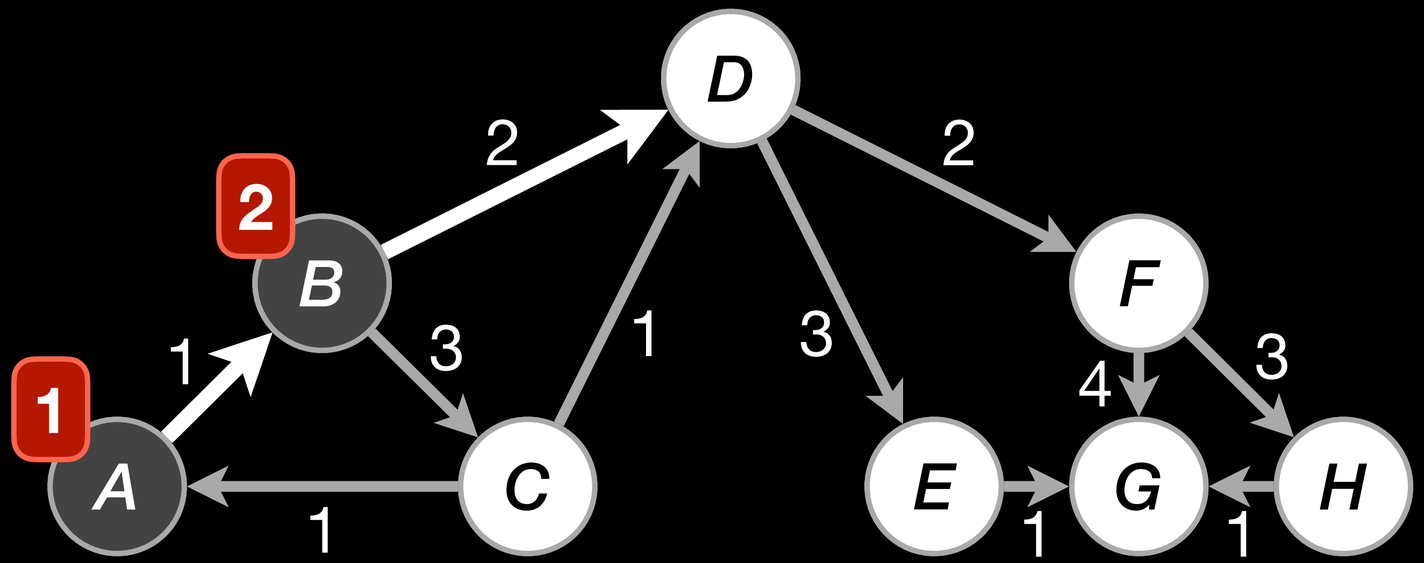
∞

C D E F G H

5. Graph Traversal　五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

(e) 使用Dijkstra算法寻找到的最短路径。



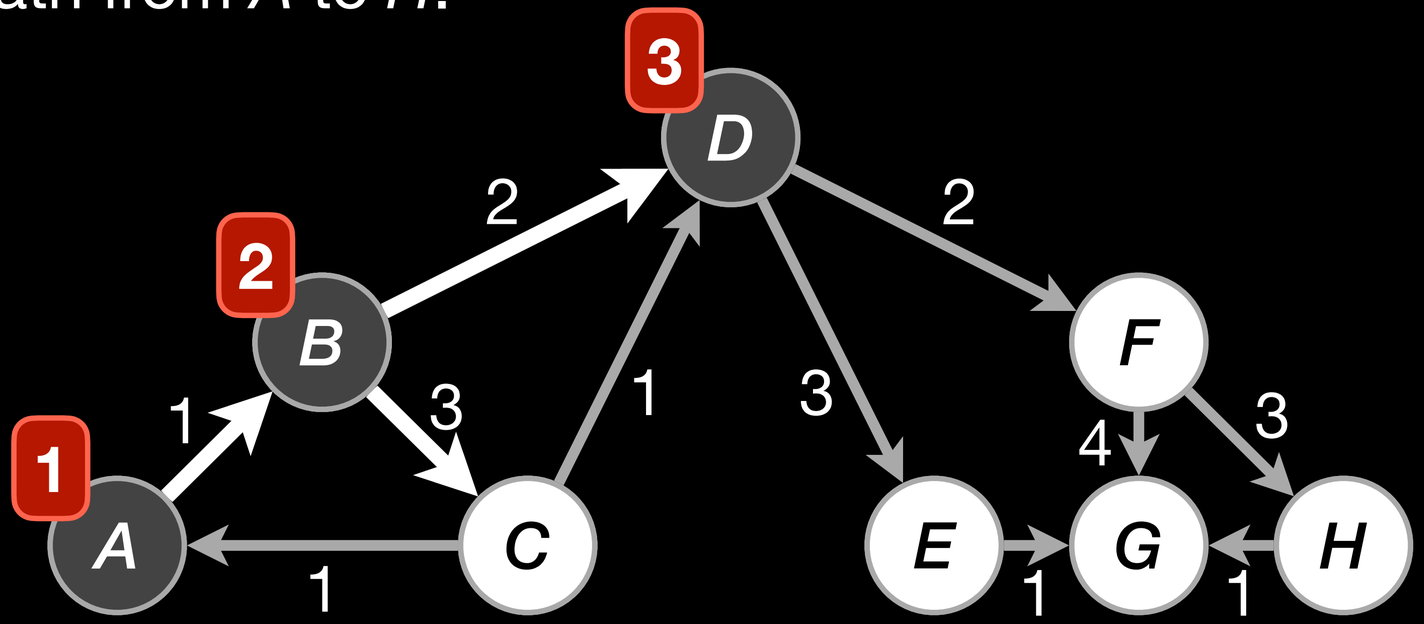
3　4　∞

D　*C*　*E F G H*

5. Graph Traversal 五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

(e) 使用Dijkstra算法寻找到的最短路径。

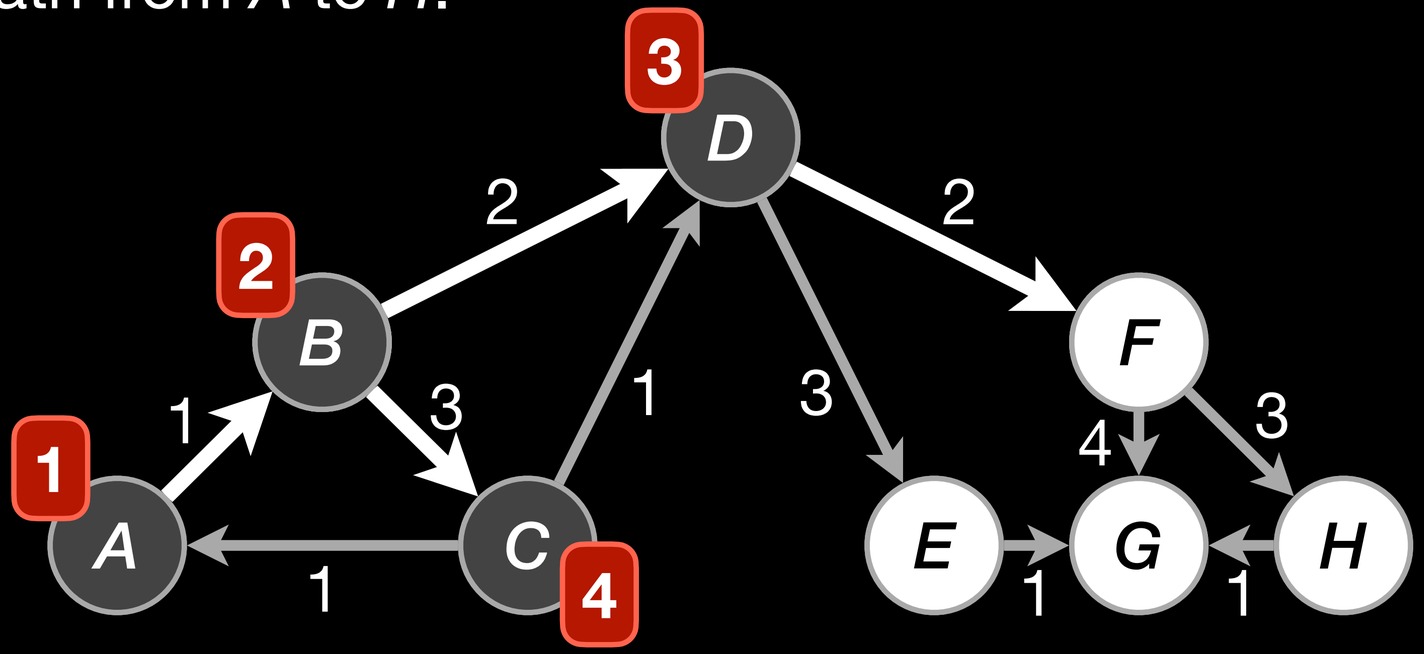


4	5	6	∞
C	*F*	*E*	*G H*

5. Graph Traversal　五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

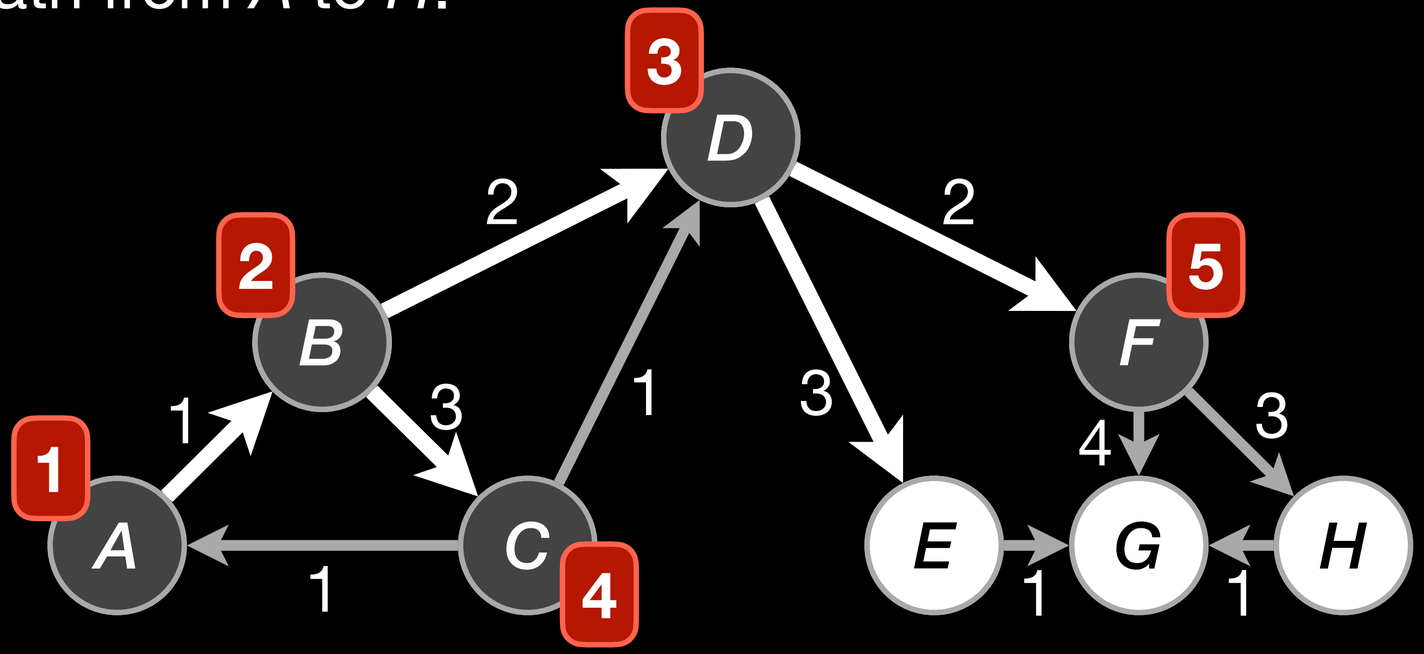
(e) 使用Dijkstra算法寻找到的最短路径。



5. Graph Traversal　五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

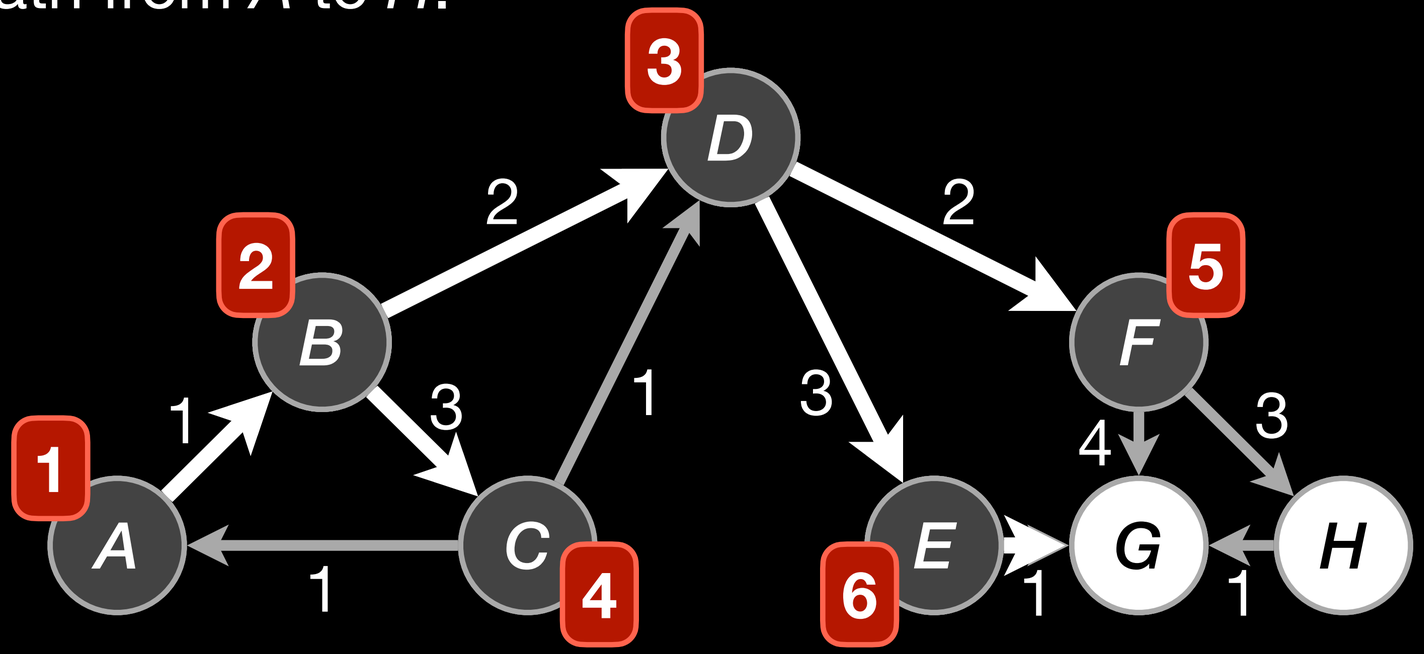
(e) 使用Dijkstra算法寻找到的最短路径。



5. Graph Traversal　五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

(e) 使用Dijkstra算法寻找到的最短路径。



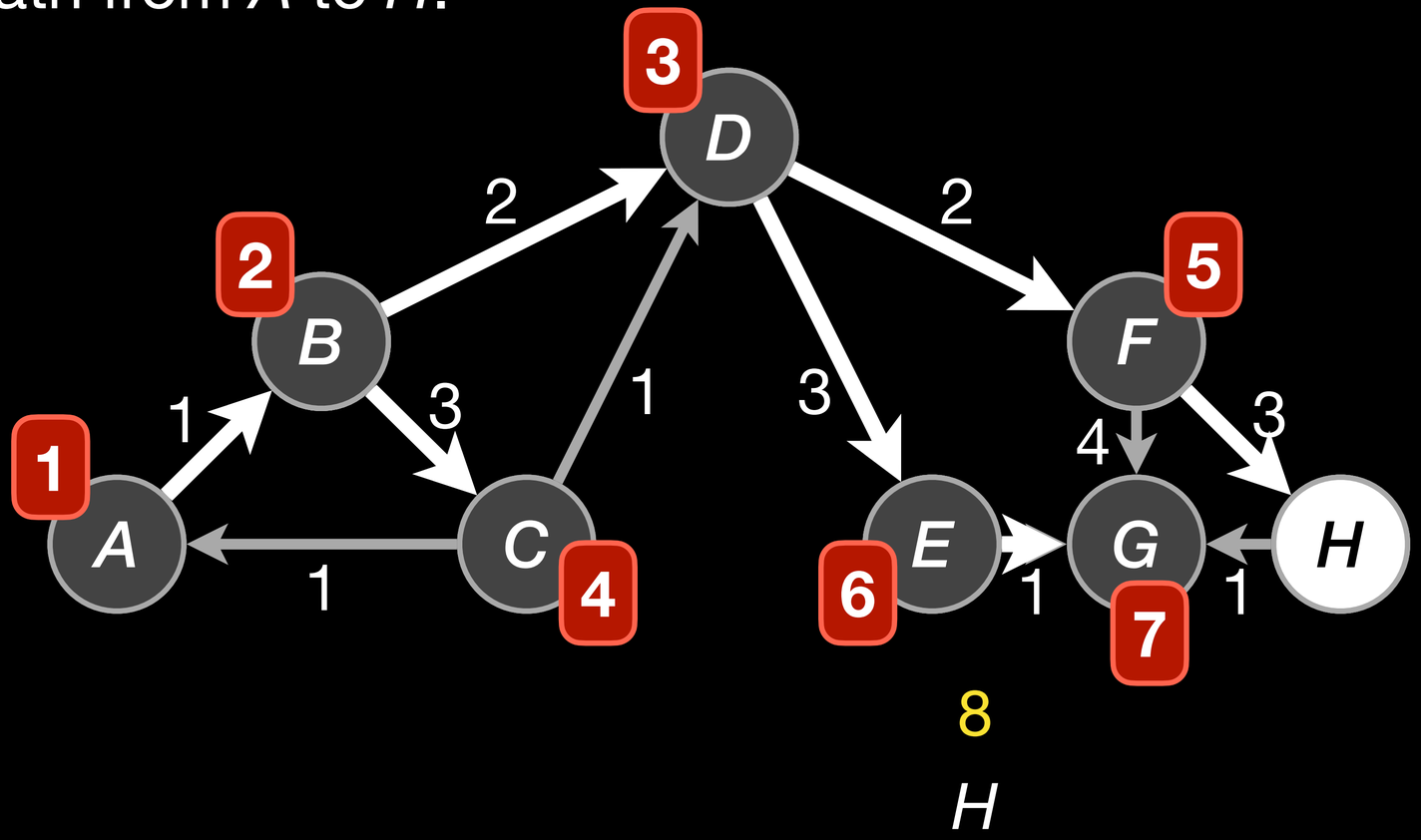
7 8

G *H*

5. Graph Traversal　五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

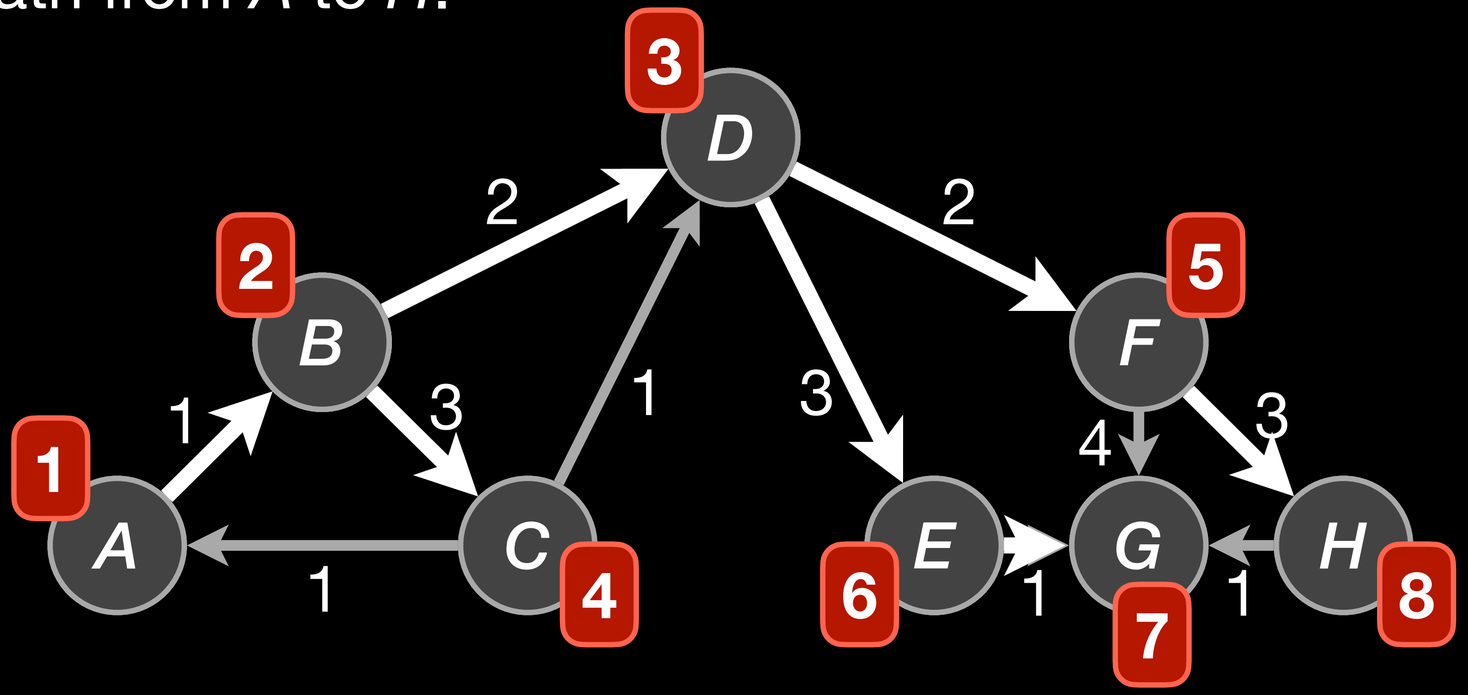
(e) 使用Dijkstra算法寻找到的最短路径。



5. Graph Traversal　五、图的遍历

(e) Use Dijkstra's algorithm to find the shortest path from *A* to *H*.

(e) 使用Dijkstra算法寻找到的最短路径。



The shortest path is: *A*→*B*→*D*→*F*→*H*.

得到的最短路径：*A*→*B*→*D*→*F*→*H*。

6. Wedding Planning

At a wedding, you need to arrange seats for guests.

(a) Suppose you know who knows which other guests. If *x* knows *y*, then *y* must know *x*. You want to arrange the guests at tables so that the guests at each table know each other or through people at their table. For example, if *x* knows *y* and *y* knows *z*, then *x*, *y*, and *z* can sit at the same table. Design an efficient algorithm that returns how many tables are needed to meet this requirement based on who knows whom. Analyze the time complexity of the algorithm.

六、婚礼计划

在婚礼上，你需要为到来的客人安排座位。

(a) 假设你知道客人之间谁互相认识。如果*x*认识*y*，则*y*必然认识*x*。你需要将客人安排到不同的桌子上，使得每个桌子的客人或者互相认识，或者通过同一个桌子的人认识。例如，如果*x*认识*y*，并且*y*认识*z*，则*x*, *y*, *z*可以坐在同一个桌子上。设计高效的算法，在给定客人之间谁互相认识的信息后，返回至少需要多少张桌子才能满足这个要求。分析该算法的时间复杂度。

6. Wedding Planning

At a wedding, you need to arrange seats for guests.

(a) Suppose you know who knows which other guests. If *x* knows *y*, then *y* must know *x*. You want to arrange the guests at tables s the guests at each table know each other through people at their table. For example, i knows *y* and *y* knows *z*, then *x*, *y*, and *z* can sit at the same table. Design an efficient algorithm that returns how many tables are needed to meet this requirement based on who knows whom. Analyze the time complexity of the algorithm.

main goal
主要目标

六、婚礼计划

在婚礼上，你需要为到来的客人安排座位。

(a) 假设你知道客人之间谁互相认识。如果*x*认识*y*，则*y*必然认识*x*。你需要将客人排到不同的桌子上，使得每个桌子的人或者互相认识，或者通过同一个桌的人认识。例如，如果*x*认识*y*，并且*y*认识*z*，则*x*, *y*, *z*可以坐在同一个桌子上。设计高效的算法，在给定客人之间谁互相认识的信息后，返回至少需要多少张桌子才能满足这个要求。分析该算法的时间复杂度。

6. Wedding Planning

At a wedding, you need to arrange seats for guests.

(a) Suppose you know who knows which other guests. If *x* knows *y*, then *y* must know *x*. You want to arrange the guests at tables so that the guests at each table know each other or through people at their table. For example, if *x* knows *y* and *y* knows *z*, then *x*, *y*, and *z* can at the same table. Design an efficient algorith that returns how many tables are needed to meet this requirement based on who knows whom. Analyze the time complexity of the algorithm.

output
输出

六、婚礼计划

在婚礼上，你需要为到来的客人安排座位。

(a) 假设你知道客人之间谁互相认识。如果*x*认识*y*，则*y*必然认识*x*。你需要将客人安排到不同的桌子上，使得每个桌子的客人或者互相认识，或者通过同一个桌子的人认识。例如，如果*x*认识*y*，并且认识*z*，则*x*, *y*, *z*可以坐在同一个桌子上。设计高效的算法，在给定客人之间谁互相认识的信息后，返回至少需要多少张桌子才能满足这个要求。分析该算法的时间复杂度。

6. Wedding Planning

At a wedding, you need to arrange seats for guests.

(a) Suppose you know who knows which other guests. If *x* knows *y*, then *y* must know *x*. You want to arrange the guests at tables so that the guests at each table know each other or through people at their table. For example, if *x* knows *y* and *y* knows *z*, then *x*, *y*, and *z* can sit at the same table. Design an efficient algorithm that returns how many tables are needed to meet this requirement based on who knows whom. Analyze the time complexity of the algorithm.

六、婚礼计划

在婚礼上，你需要为到来的客人安排座位。

(a) 假设你知道客人之间谁互相认识。如果*x*认识*y*，则*y*必然认识*x*。你需要将客人安排到不同的桌子上，使得每个桌子的客人或者互相认识，或者通过同一个桌子的人认识。例如，如果*x*认识*y*，并且*y*认识*z*，则*x*, *y*, *z*可以坐在同一个桌子上。设计高效的算法，在给定客人之间谁互相认识的信息后，返回至少需要多少张桌子才能满足这个要求。分析该算法的时间复杂度。

6. Wedding Planning

At a wedding, you need to arrange seats for guests.

(a) Suppose you know who knows which other guests. If *x* knows *y*, then *y* must know *x*. You want to arrange the guests at tables so that the guests at each table know each other or through people at their table. For example, if *x* knows *y* and *y* knows *z*, then *x*, *y*, and *z* can be at the same table. Design an efficient algorithm that returns how many tables are needed to meet this requirement based on who knows whom. Analyze the time complexity of the algorithm.

input: an undirected graph

输入：无向图

六、婚礼计划

在婚礼上，你需要为到来的客人安排座位。

(a) 假设你知道客人之间谁互相认识。如果*x*认识*y*，则*y*必然认识*x*。你需要将客人安排到不同的桌子上，使得每个桌子的客人或者互相认识，或者通过同一个桌的人认识。例如，如果*x*认识*y*，并且*y*认识*z*，则*x*, *y*, *z*可以坐在同一个桌子。设计一个高效的算法，在给定客人之间谁认识的信息后，返回至少需要多少张桌子才能满足这个要求。分析该算法的时间复杂度。

6. Wedding Planning

六、 婚礼计划

Design an algorithm:

Input:
an undirected graph indicating
who knows whom

Output:
the minimum number of tables such that
people at every table know each other =
the number of connected components

设计算法：

输入：
表示谁认识谁的无向图

输出：
至少需要多少张桌子，这样
每个桌子上的客人互相认识 =
连接组件的数量

6. Wedding Planning

六、 婚礼计划

Design an algorithm that counts
the number of connected components
in an undirected graph.

设计一个计算无向图中
连接组件数量的算法。

Simple solution:
Run an algorithm that finds the connected components (e.g. the one from the book), and only output the number of components found.

简单解决方案：
运行查找连接组件的算法（例如，书本中的组件），只输出找到的组件数。

6. Wedding Planning

六、 婚礼计划

Design an algorithm that counts
the number of connected components
in an undirected graph.

设计一个计算无向图中
连接组件数量的算法。

More advanced solution:
In an undirected graph, DFS-VISIT() of any vertex will make all vertices that are in the same connected component black.
Repeatedly call DFS-VISIT() until no more white vertices are left, similar to DFS().

更高级的解决方案：
在无向图中，任何顶点的DFS-VISIT()
将使同一连接组件中的所有顶点
变为黑色。
重复调用DFS-VISIT()，
直到不再剩下白色顶点，类似于DFS()。

6. Wedding Planning　六、婚礼计划

Design an algorithm that counts
the num
in an ur

More a
In an ur
vertex v
same c
Repeat
white ve

设计一个计算无向图中

SIT()

DFS()。

```
COUNT-CONNECTED-COMPONENTS(G)
for each vertex u ∈ G.V
        u.color = white, u.π = NIL
number-of-components = 0
for each vertex u ∈ G.V
        if u.color == white
                DFS-VISIT(G,u)
                number-of-components = number-of-components + 1
return number-of-components
```

6. Wedding Planning　六、婚礼计划

Design an algorithm that counts
the number of connected components
in an undire

More adva
In an undire
vertex will n
same conne
Repeatedly call DFS-Visit() until no more
white vertices are left, similar to DFS().

设计一个计算无向图中
连接组件数量的算法

'ISIT()
i

重复调用DFS-Visit()，
直到不再剩下白色顶点，类似于DFS()。

COUNT-CONNECTED-COMPONENTS(*G*)
Initialize the data structures as for DFS().
Set the number of components to 0.
While there are vertices that have not yet been visited,
pick one, call DFS-VISIT() of this vertex,
and increase the number of components by 1.
The final number of components is the result of the algorithm.

6. Wedding Planning

Two things remain to be done.

- **Time complexity**
  Either algorithm has time complexity $O(|V| + |E|)$ (the complexity of DFS or of searching connected components).

- **How to use the algorithm**
  Construct a graph where vertices are guests and edges exist between guests that know each other.
  Apply the algorithm to this graph.

六、婚礼计划

还有两件事。

- **时间复杂性**
  两种算法都有时间复杂度 $O(|V| + |E|)$（深度优先搜索的或搜索连接组件的复杂度）。

- **如何使用该算法**
  构建一个图，其中顶点是客人，边存在于相互认识的客人之间。
  将算法应用于此图。

6. Wedding Planning　六、婚礼计划

6. Wedding Planning

Two things remain to be done.

- **Time complexity**
  Either algorithm has time complexity $O(|V| + |E|)$ (the complexity of DFS or of searching connected components).

- **How to use the algorithm**
  Construct a graph where vertices are guests and edges exist between guests that know each other.
  Apply the algorithm to this graph.

六、婚礼计划

还有两件事。

- **时间复杂性**
  两种算法都有时间复杂度 $O(|V| + |E|)$（深度优先搜索的或搜索连接组件的复杂度）。

- **如何使用该算法**
  构建一个图，其中顶点是客人，边存在于相互认识的客人之间。
  将算法应用于此图。

6. Wedding Planning

六、婚礼计划

(a) Now consider another scenario: There are only two tables and some guests dislike each other. If *u* dislikes *v*, then *v* must dislike *u*. Your goal is to allocate guests so that no two guests who dislike each other share their table. For example, if the dislike relationship between guests is as in (a), *A* and *C* can sit at one table and *B*, *D* and *E* at the other table. If the dislike relationship between guests is as in (b), this goal cannot be achieved. Design an efficient algorithm that returns whether the goal can be achieved with a given dislike relationship between guests. Analyze the time complexity of the algorithm.

(b) 现在考虑另一个场景：假设只有两张桌子，另外你知道有些客人不喜欢对方。如果*u*不喜欢*v*，则必然*v*不喜欢*u*。你的目标是将客人分配到两张桌子上，使得同一桌子上不存在两个客人不喜欢对方。例如，如果客人间不喜欢的关系如图(a)所示，则可以将*A*, *C*安排在一个桌子，*B*, *D*, *E*安排在另一个桌子。如果客人间的不喜欢关系如图(b)所示，则这个目标无法达到。设计高效的算法，在给定客人间的不喜欢关系之后，返回目标是否可能达到。分析算法的时间复杂度。

6. Wedding Planning

六、婚礼计划

Design an algorithm:

Input:
an undirected graph indicating
who dislikes whom

Output:
“Yes” or “No” to answer the question:
Is it possible to seat guests so that no guests who dislike each other share their table?

设计算法：

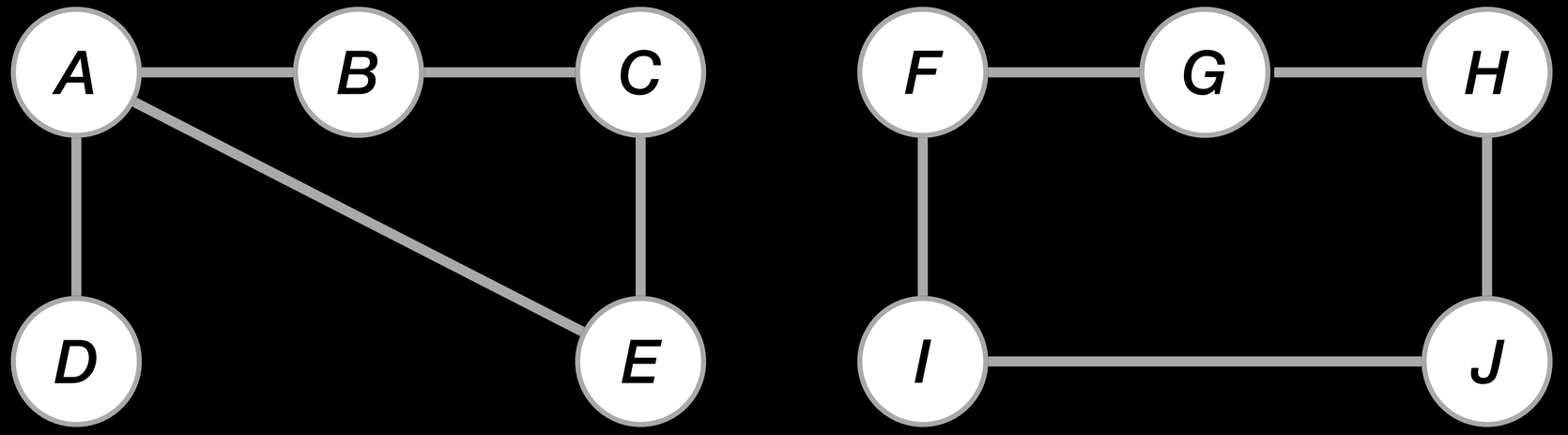
输入：
表示谁不喜欢谁的无向图

输出：
“是”或者“否”回答问题：
可不可以将客人分配到两张桌子上，使得同一桌子上
不存在两个客人不喜欢对方?

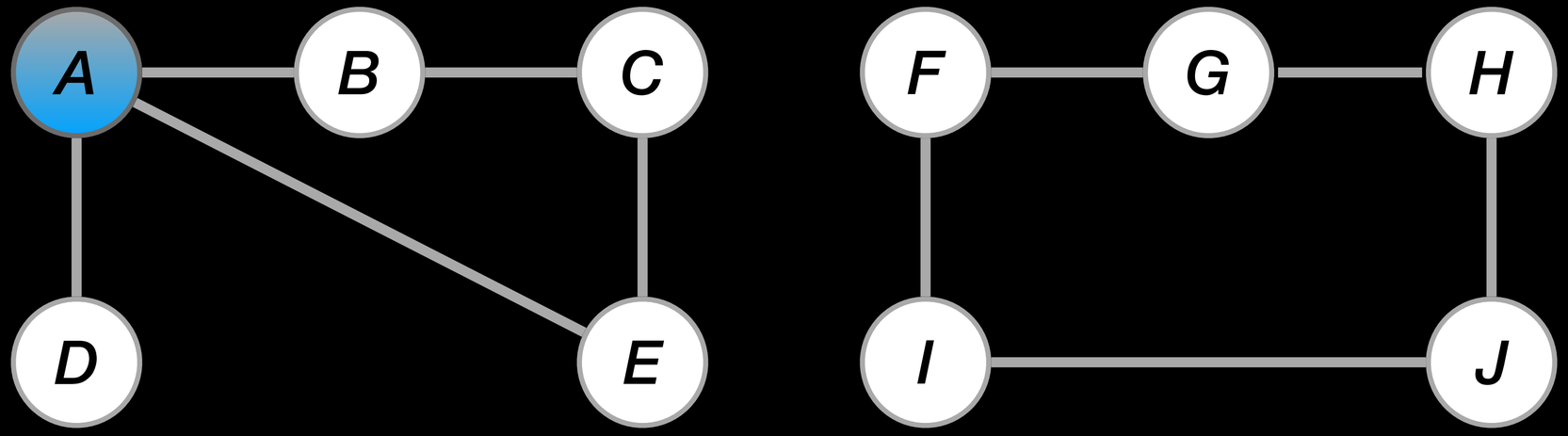
6. Wedding Planning 六、婚礼计划

- I don’t think we discussed an algorithm that applies here directly. However, I can think of a procedure similar to a search algorithm that also distributes the vertices over the tables.

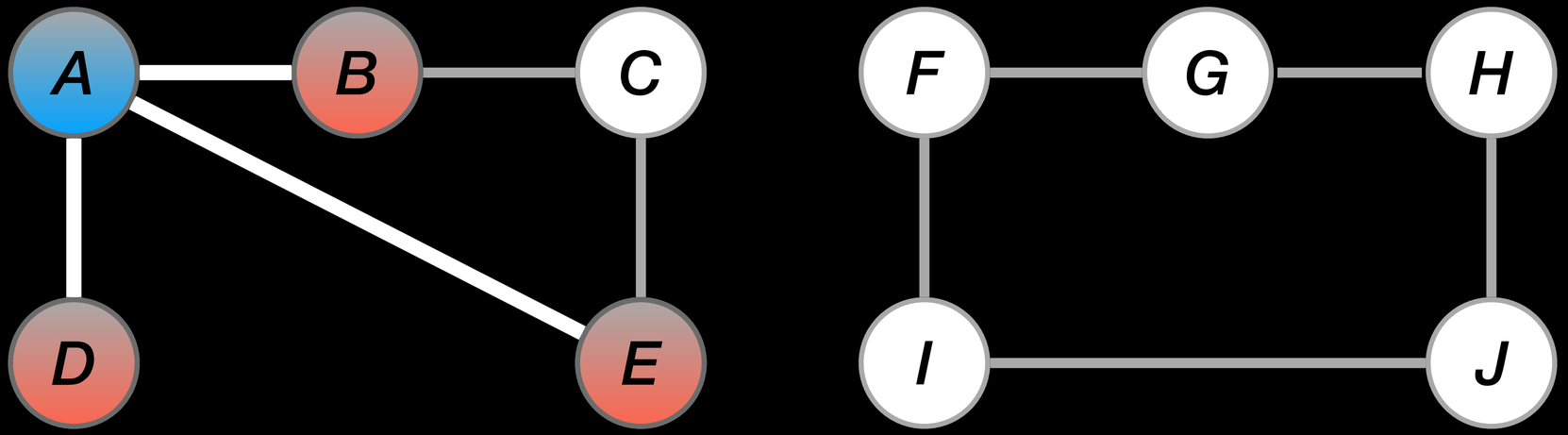
6. Wedding Planning 六、 婚礼计划



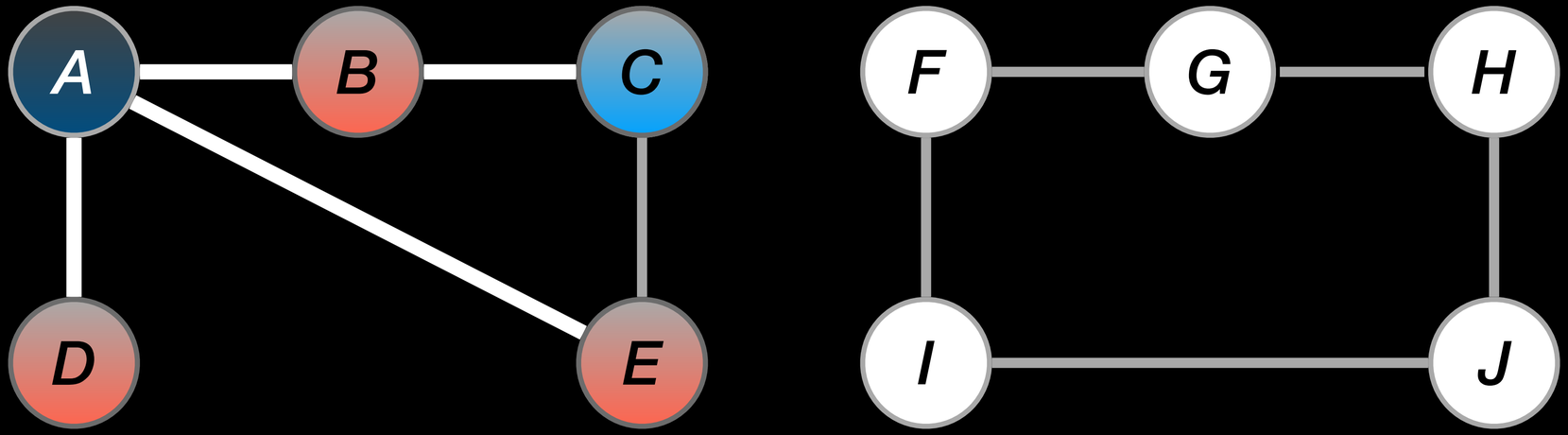
6. Wedding Planning 六、婚礼计划



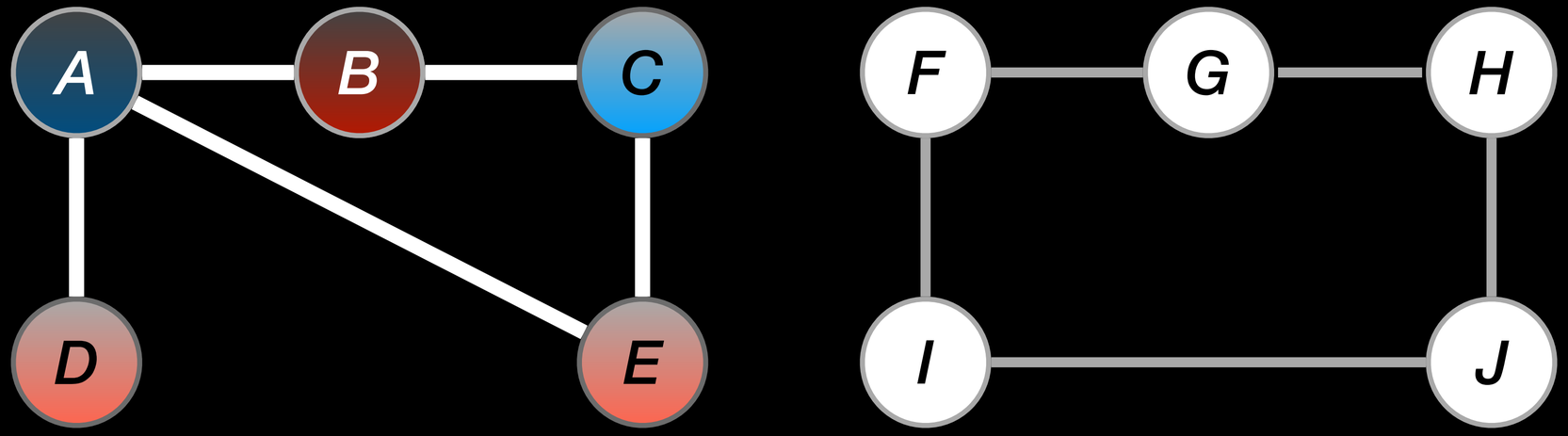
6. Wedding Planning 六、婚礼计划



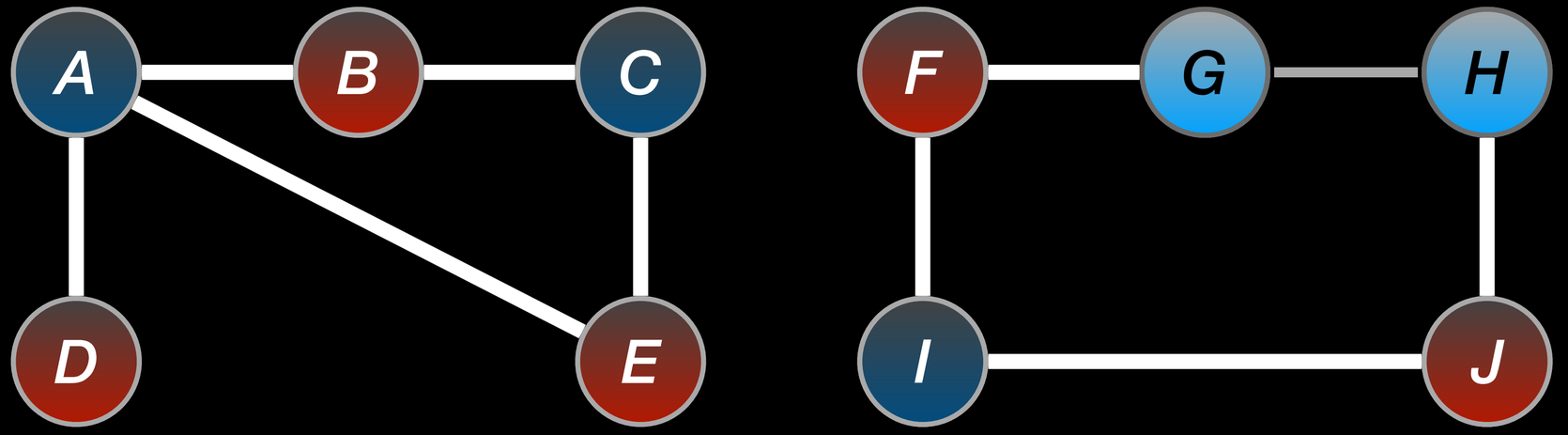
6. Wedding Planning 六、婚礼计划



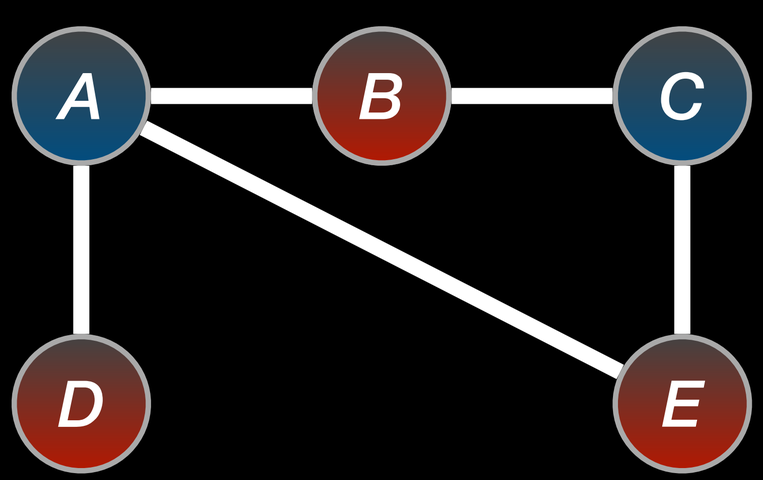
6. Wedding Planning 六、婚礼计划

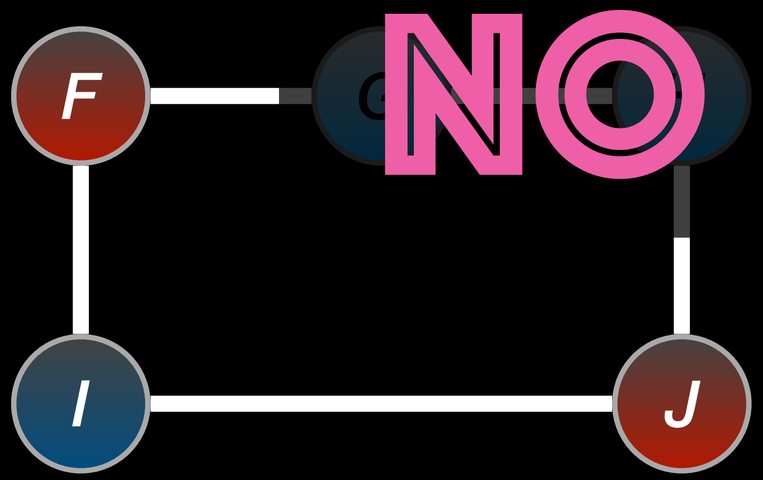


6. Wedding Planning 六、婚礼计划



6. Wedding Planning 六、婚礼计划





6. Wedding Planning　六、婚礼计划

CHECK-SEATING-ORDER(G)
Initialize the data structures as for BFS.
Pick any vertex $s \in G.V$ that is not yet assigned a table,
assign it a table (e.g. the blue table),
and run BFS with s as source.
While running a BFS search through the vertices connected with s,
every time a new vertex is discovered place it at the correct table
(i.e. at the table different from its neighbor).
Every time a vertex that was discovered earlier is reached,
check whether it is placed at the correct table.
When the check fails, immediately return “No”.

After all vertices have been assigned a table
and no check has failed, return “Yes”.

6. W

```
CHECK-SEATING-ORDER(G)
for each vertex u ∈ G.V
        u.color = white, u.table = none
while there is some white vertex s ∈ G.V
        s.color = gray, s.table = blue
        Q = empty queue
        ENQUEUE(Q,s)
        while Q is not empty
                u = DEQUEUE(Q)
                for each v adjacent to u
                        if v.color == white
                                v.color = gray, v.table = the table different from u.table
                                ENQUEUE(Q,v)
                        else if v.table == u.table
                                return “No”
                u.color = black
return “Yes”
```

6. Wedding Planning

六、 婚礼计划

- **Correctness**
  If the algorithm finds a seating order, guests can obviously be seated.
  If the algorithm finds a problem, it is caused by a cycle with an odd number of vertices, and in this situation it is impossible to find a suitable seating order.

- **正确性**
  如果算法找到了席次，
  客人显然可以就座。
  如果算法发现一个问题，
  是由奇数个顶点的循环引起的，
  在这种情况下不可能找到合适的席次。

- (I do not expect a full correctness proof but a proof idea for a new algorithm.)

- （不希望有一个完整的正确性证明，
  但希望有一个新算法的证明想法。）

6. Wedding Planning

- **Time complexity**
  The time complexity is $O(|V| + |E|)$,
  as for BFS.
  This is optimal, as we have to find a table for every guest $O(|V|)$, and have to check every dislike at least once $O(|E|)$.

六、 婚礼计划

- **时间复杂性**
  时间复杂度为$O(|V| + |E|)$,
  至于广度优先搜索。
  这是最佳选择，因为我们必须为每位客人找到一张桌子 $O(|V|)$,
  并且必须至少检查一次每个不喜欢的边 $O(|E|)$。