基本图算法

A picture is worth
a thousand words

引言

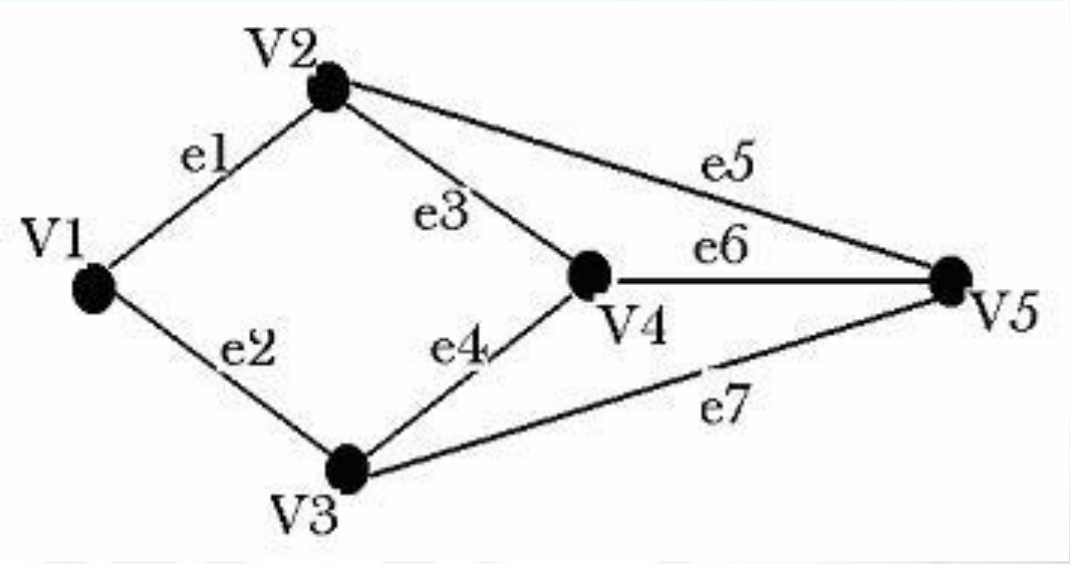
- 在前一节中，我们已经学习了几种常用的经典数据结构
- 树是我们学习过的唯一一种能够表示非线性的层级关系的数据结构
- 树的形式有一定的约束，即每个节点至多只能有一个父节点，但可以有多个子节点
- 如果去掉这种约束，就得到了在这一节将要介绍的另一种数据结构——图（graph）

概要

- 这一节我们将主要学习关于图的以下内容
  - 图的表示
  - 深度优先搜索
  - 广度优先搜索
  - 拓扑排序

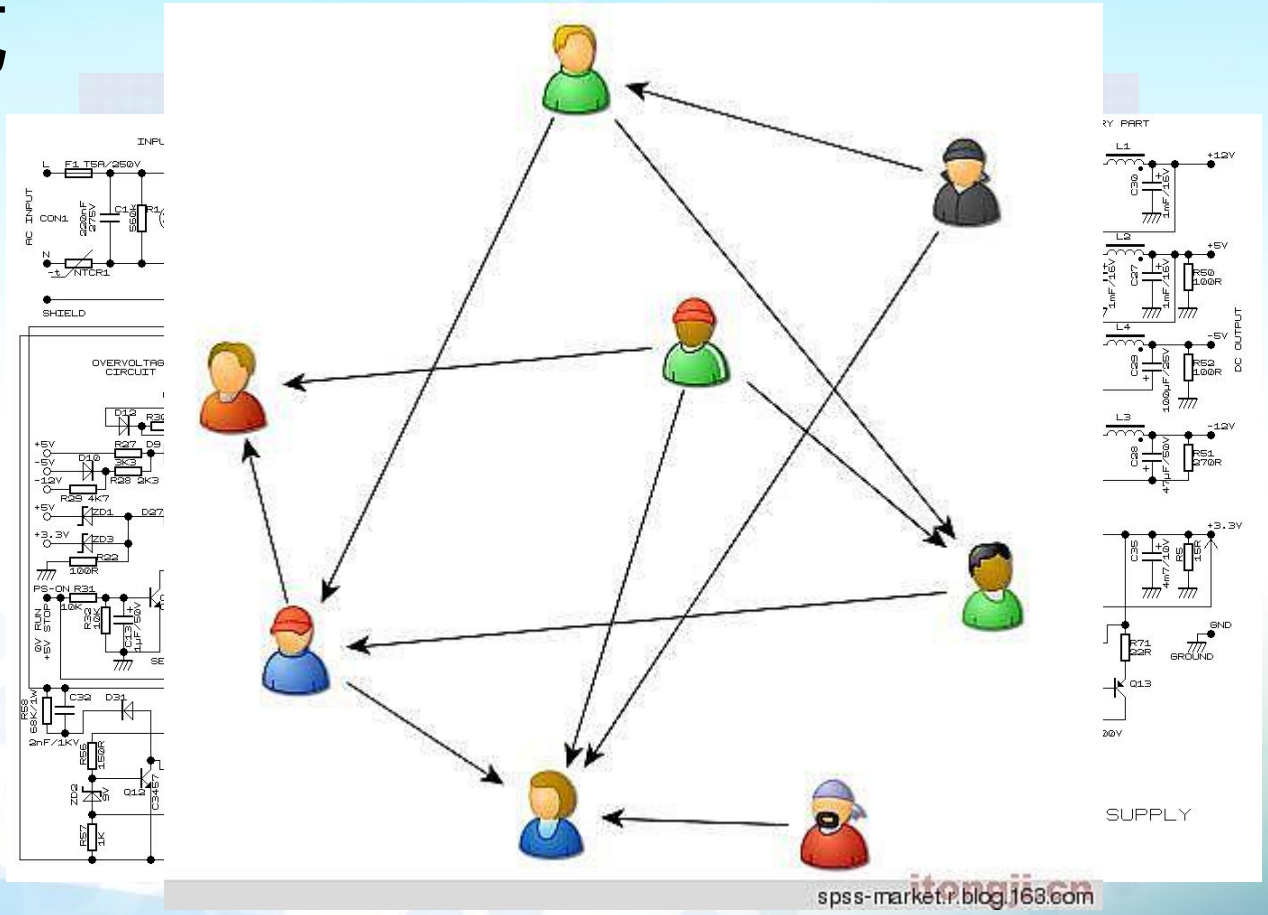
图

- 图不再受像树那样只能有一个父节点的约束，图的每一个顶点可以与多个其它顶点相关联，各顶点之间的关系是任意的



图

- 生活中有许多事物都可以抽象为图的表示方式



图的形式定义

- 图是由顶点集合（vertex）及顶点间的关系集合组成的一种数据结构：

$$Graph = (V, E)$$

- 其中
  - V是顶点集合
  - $E = \{(x, y)|x, y \in V\}$是顶点间的关系集合，也称为边（edge）集合

图的分类

- 图可以分为有向图（directed graph）和无向图（undirected graph）
- 在有向图中，顶点对<*x*, *y*>是有序的，称为从*x*到*y*的一条有向边，这里<*x*, *y*>与<*y*, *x*>是不同的两条边，就像微博粉丝，微信公众号
- 在无向图中，顶点对(*x*, *y*)是无序的，(*x*, *y*)和(*y*, *x*)是同一条边，就像人人或是微信上的好友关系

关于图的一些概念

- 我们这里讨论的图都是简单图，即认为顶点没有与自己相连的边（自环），也没有顶点之间有多条边直接相连（多重图）
- 如果(x, y)是$E(G)$中的一条边，那么我们称x与y互为邻接顶点（adjacent vertex）
- 与顶点x关联的边数成为x的度（degree），记为deg(x)
  - 如果是有向图则具体分为入度indeg(x)和出度outdeg(x)

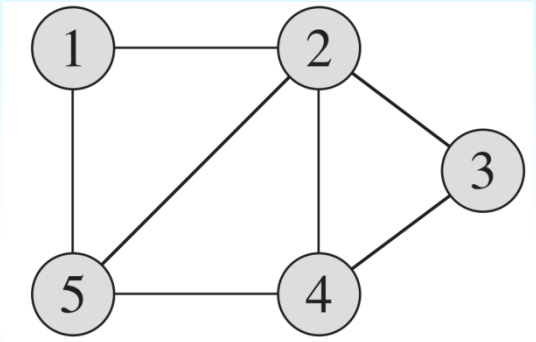
图的表示

- 表示一个图，最重要的就是描述每个顶点的邻接关系
- 一般来说，表示一个图$G=(V, E)$有两种标准形式：
  - 邻接表（adjacency lists）
  - 邻接矩阵（adjacency matrix）
- 其中邻接表的表示方法由于其紧凑性而较为常用

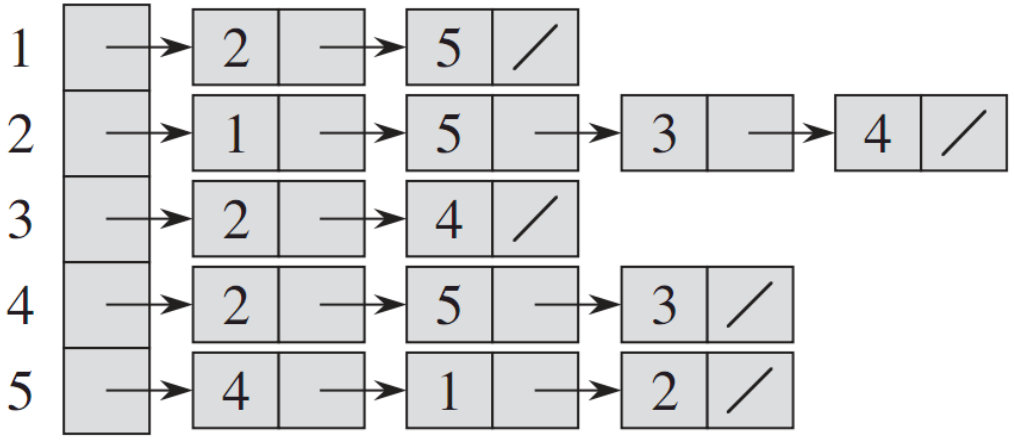
图的邻接表表示

- 图$G=(V, E)$的邻接表表示是由$|V|$个线性表组成的，V中的每个顶点都对应一个线性表
- 对于每个顶点$u \in V$，其对应的线性表储存了所有满足$(u, v) \in E$的顶点v，即包含了u在G中所有的邻接顶点
- 在没有特殊要求的情况下，邻接表中的顶点通常是按任意顺序排列储存的

图的邻接表表示

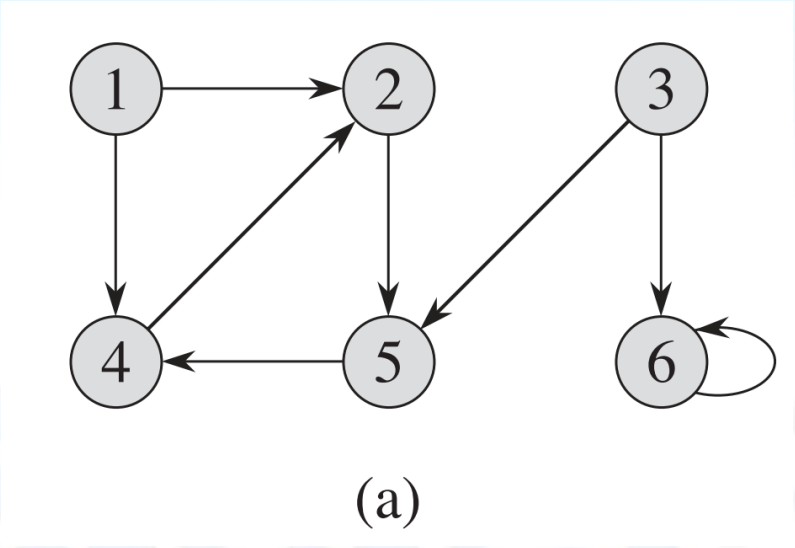


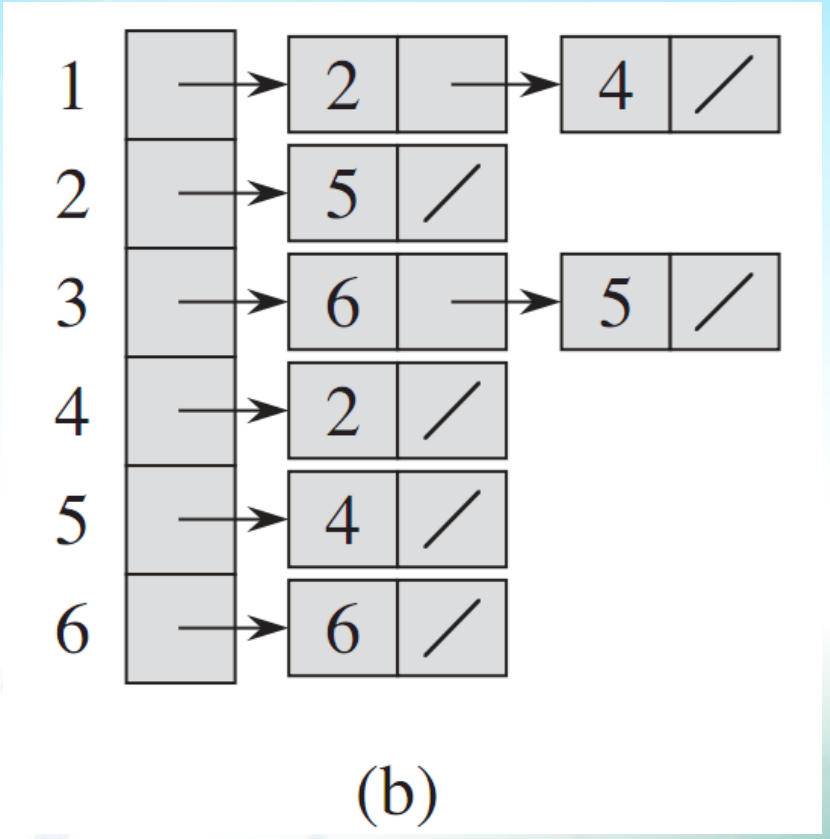
(a)



(b)

图的邻接表表示



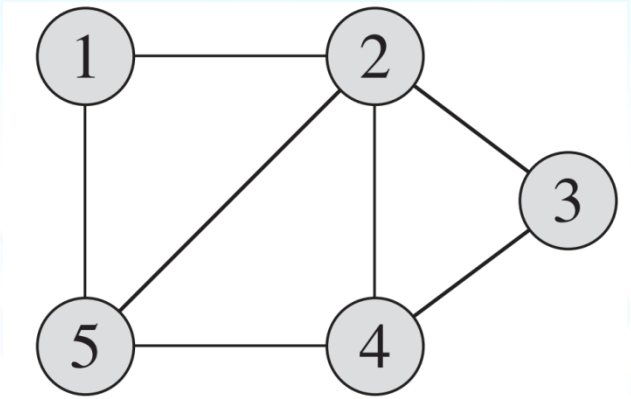


图的邻接矩阵表示

- 对于图$G=(V, E)$，我们假定顶点按某个顺序被编号为1，2，……，$|V|$，从而图G的邻接矩阵表示即是一个$|V|\times|V|$的矩阵$A=(a_{ij})$，其矩阵项满足

$$a_{ij} = \begin{cases} 1 \; if \; (i,j) \in E \\ 0 \; otherwise \end{cases}$$

图的邻接矩阵表示

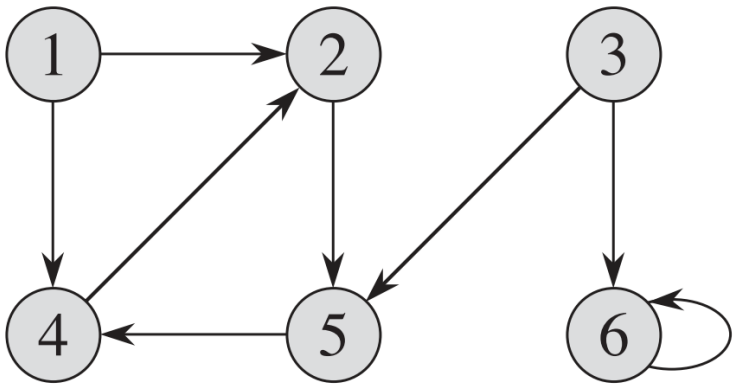


(a)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

图的邻接矩阵表示



(a)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

表示方法的比较

- 邻接表的表示方法更为节省空间，通常只需要$|E|$个储存单元来保存相应的$|E|$条边即可，而邻接矩阵的表示方法需要$|V|^2$个储存单元
- 在图的边很稠密的情况下，两种表示方法占用的空间差别不大
- 邻接矩阵的表示方法可以快速查询某两个顶点之间是否有边相连

图的遍历

- 图的遍历指的是给定图*G*和其中任意一个顶点*s*，从*s*出发，沿着图中的边访问图中所有的顶点，且每个顶点仅被访问一次
- 这里所说的“访问”，根据具体的应用可能有不同的含义，如输出顶点信息，或是修改顶点的某个属性等等
- 图的遍历是我们处理和解决一些与图相关的应用问题的先决条件

图的遍历

- 绝大多数有关图的问题都涉及图的整体结构，因此只有在收集到所有顶点和边的信息之后，才能正确地求解问题
- 另一方面，如果不能控制顶点的访问次数，算法的冗余计算将使问题复杂化而导致时间效率低下
- 基本的图遍历算法包括了深度优先搜索和广度优先搜索

深度优先搜索

- 深度优先搜索（depth first search）是一个不断探查和回退的过程
- 在探查的每一步开始之前，算法都有一个当前顶点（最开始即是起始顶点）
- 每一步探查中，我们在当前顶点v的所有邻接顶点中，找出尚未访问过的一个，将其作为下一步探查的当前顶点，即我们永远希望向着更“深”的层次去探索

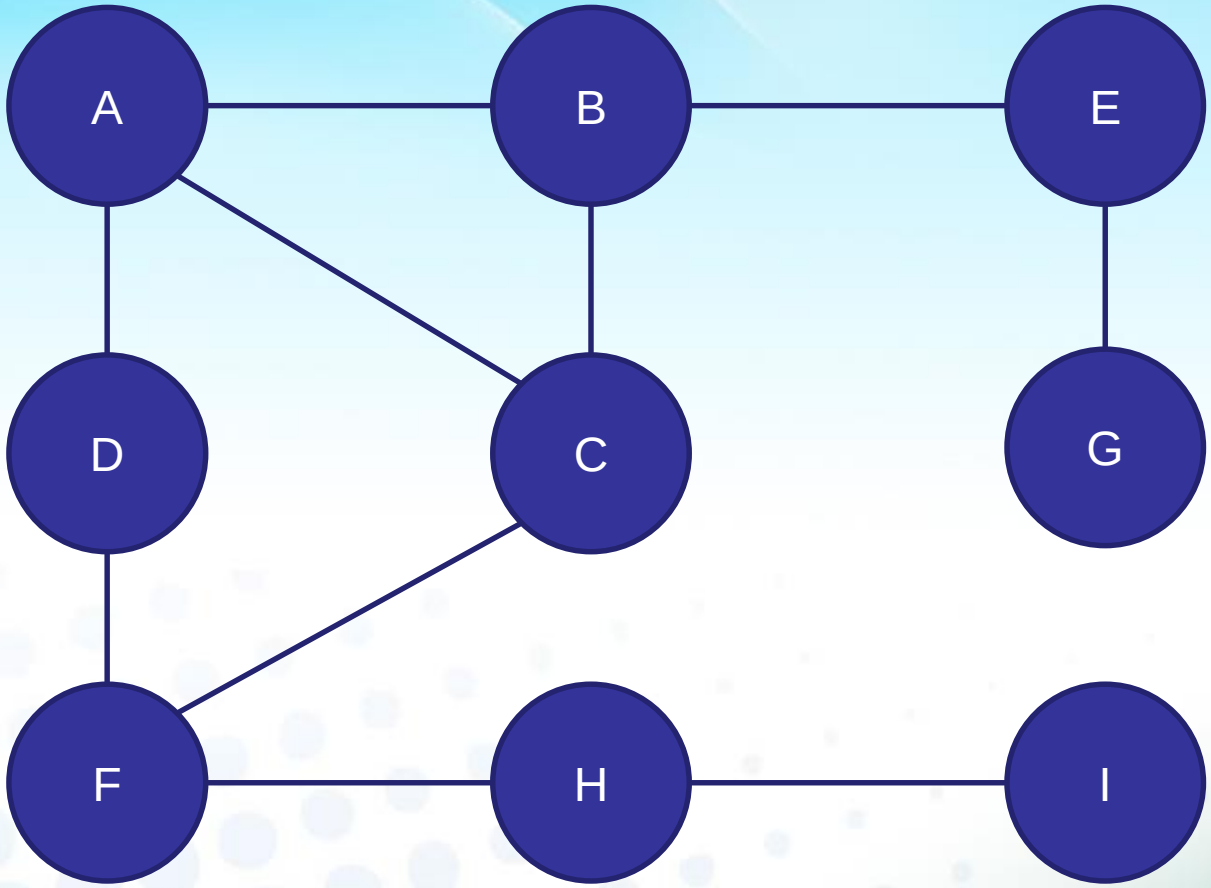
深度优先搜索

- 倘若当前顶点的所有邻接顶点都已经被访问过，则退回一步，将前一步所访问的顶点重新取出，当作探查的当前顶点
- 重复以上过程，直到所有起始顶点的邻接顶点都已经被访问到，这也就表明所有从起始顶点可到达的顶点都已经被访问过了
- 但仍然可能有未被访问的顶点，我们在它们中间随机选择一个再作为当前顶点，继续重复以上过程，直到所有顶点都被访问

深度优先搜索

- 在这个过程中，为了避免重复访问已经探查过的顶点，我们一般将已经被访问过的顶点作上标记，在探查之前先检查该标记，确定未访问过之后再进行探查
- 针对深度优先搜索这种不断向更“深”处探查，没有边之后再回退的过程，你觉得使用我们学过的哪一种数据结构进行模拟比较合适？

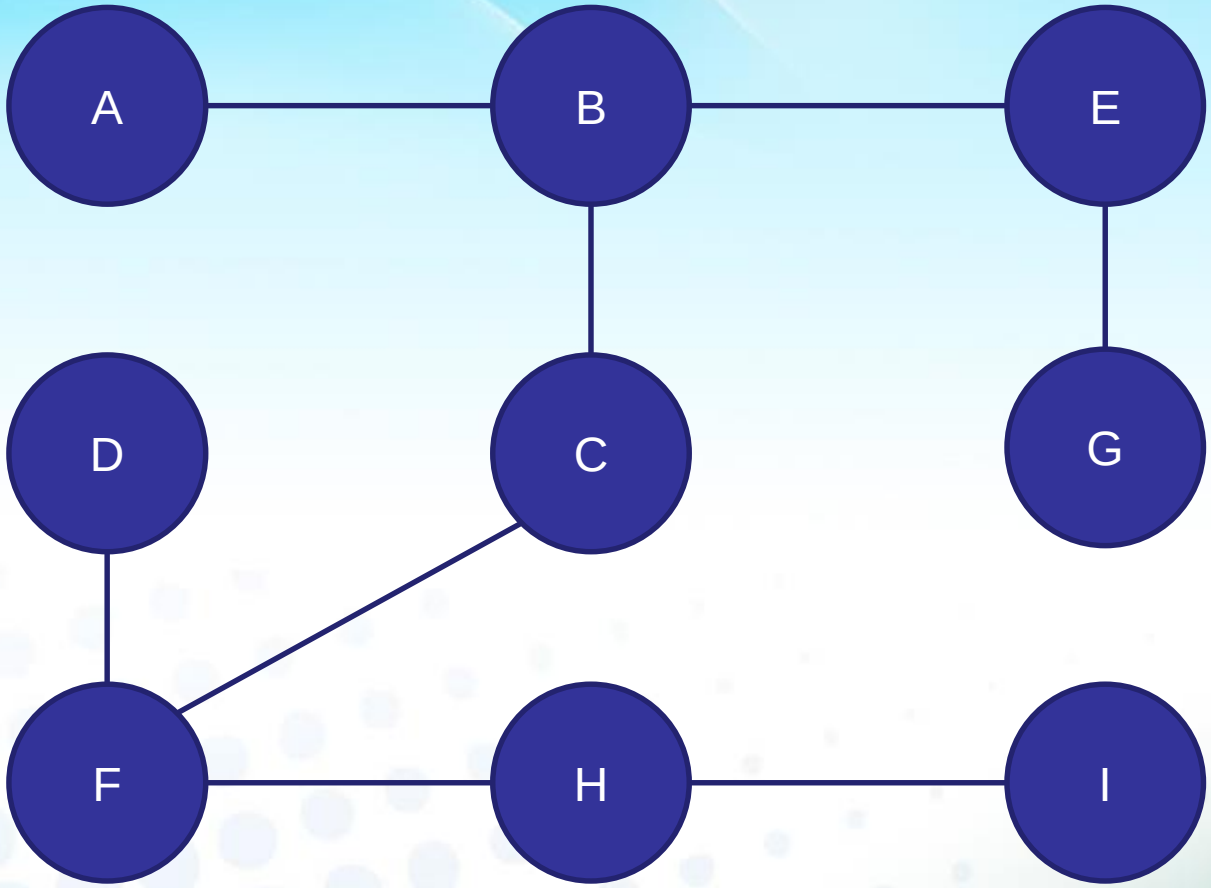
深度优先搜索



深度优先搜索

- 上图给出了一个深度优先搜索的实例，从顶点A出发做深度优先搜索，可以遍历该图的所有顶点
- 在深度优先搜索的过程中，对于所有访问过的顶点和经过的边，由于我们每个顶点仅访问一次，因此它们构成了一棵树
- 这棵树称之为图的深度优先生成树（DFS tree），在许多问题当中有重要的应用

深度优先生成树



深度优先搜索的伪代码

DFS(*G*, *s*)
for each vertex *u* in *V*[*G*] **do**
    visited[*u*] ← false
DFS-Visit(*s*)
for each vertex *u* in *V*[*G*] **do**
    if (not *visited*[*u*])
        then DFS-Visit(*u*)

深度优先搜索

- 我们前面已经提到，深度优先搜索的过程可以使用栈来模拟，当然也可以使用递归的形式来完成
- 这里实现的是深度优先搜索最基本的框架，在具体应用中，可能还会记录每个结点是由哪个结点扩展得到的，以及结点的探查起始时间和探查结束时间等信息

递归实现

DFS-Visit(*u*)

visited[*u*] ← true

for each *v* in the adjacency list of *u* **do**

    if (not *visited*[*v*])

        then DFS-Visit(*v*)

非递归栈实现

DFS-Visit(*u*)
visited[*u*] ← true
Push(*Stack*, *u*)
while *Stack* is not empty **do**
    v ← next unvisited adjacent vertex of *u*
    if (such *v* exist)
        then Push(*Stack*, *v*)
            visited[*v*] ← true
            u ← *v*
        else *u* ← Pop(*Stack*)

广度优先搜索

- 与深度优先搜索不同，广度优先搜索（breadth first search）没有探查和回退的过程，而是一个逐层遍历的过程
- 从起始点开始作为首层，然后对每层的所有顶点，都向外扩展访问那些未被访问过的邻接顶点，而这些扩展出来的顶点就作为下一层的顶点
- 依此类推，直到所有顶点都被访问为止

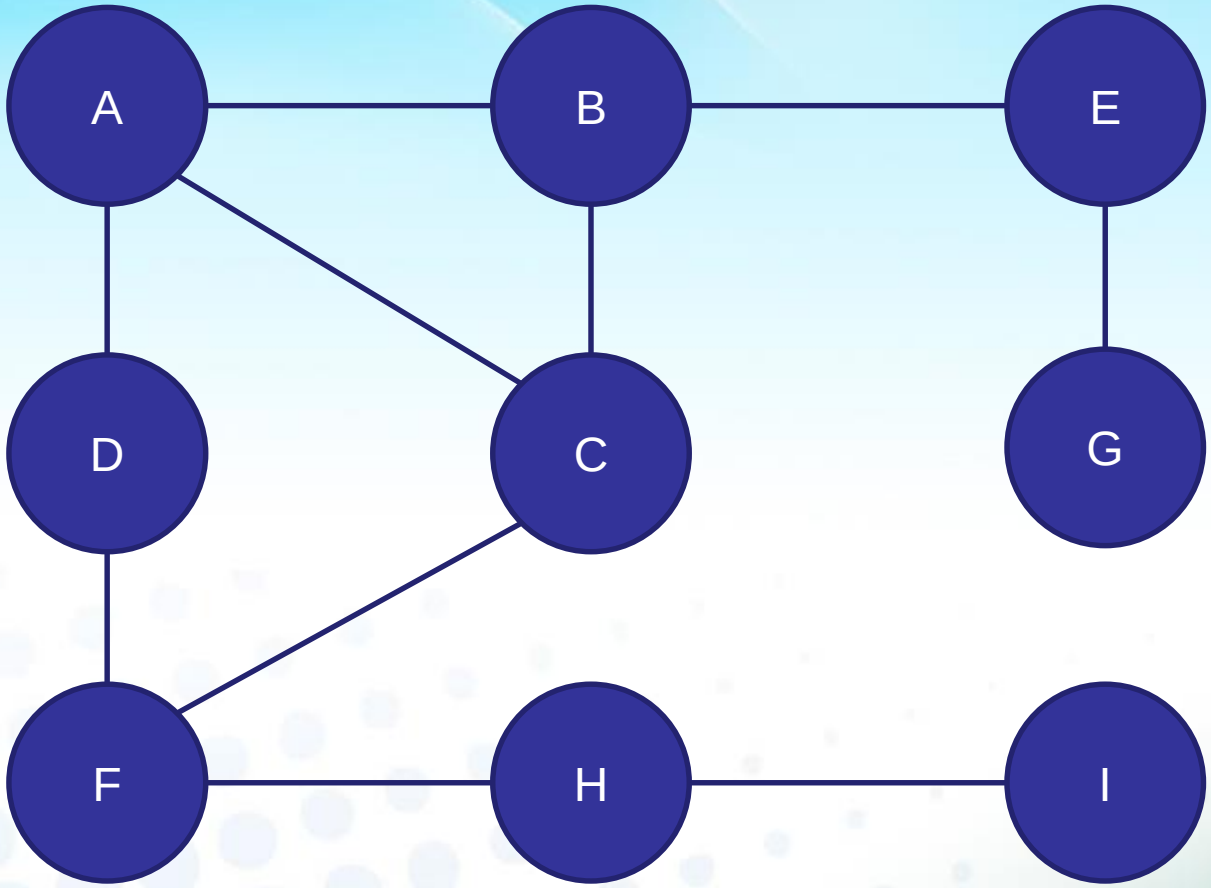
广度优先搜索

- 该遍历算法之所以称为广度优先搜索，是因为它始终是将已访问顶点和未访问顶点的边界，沿其广度方向向外扩展
- 广度优先搜索还能用来计算起始点到所有可达顶点之间的距离（即最少的边数），因为我们每次扩展都探索了周围所有可能的顶点，总是首先发现和起始点s距离为k的顶点，然后才会发现和s距离为k+1的顶点

广度优先搜索

- 与深度优先搜索类似，广度优先搜索在遍历过程中，为了避免重复访问，同样需要标记那些已访问过的结点
- 与深度优先搜索不同的是，广度优先搜索不是一个递归的结构，其算法也不是探查与回退的过程
- 针对广度优先搜索这种逐层访问的过程，你觉得用什么数据结构模拟比较合适？

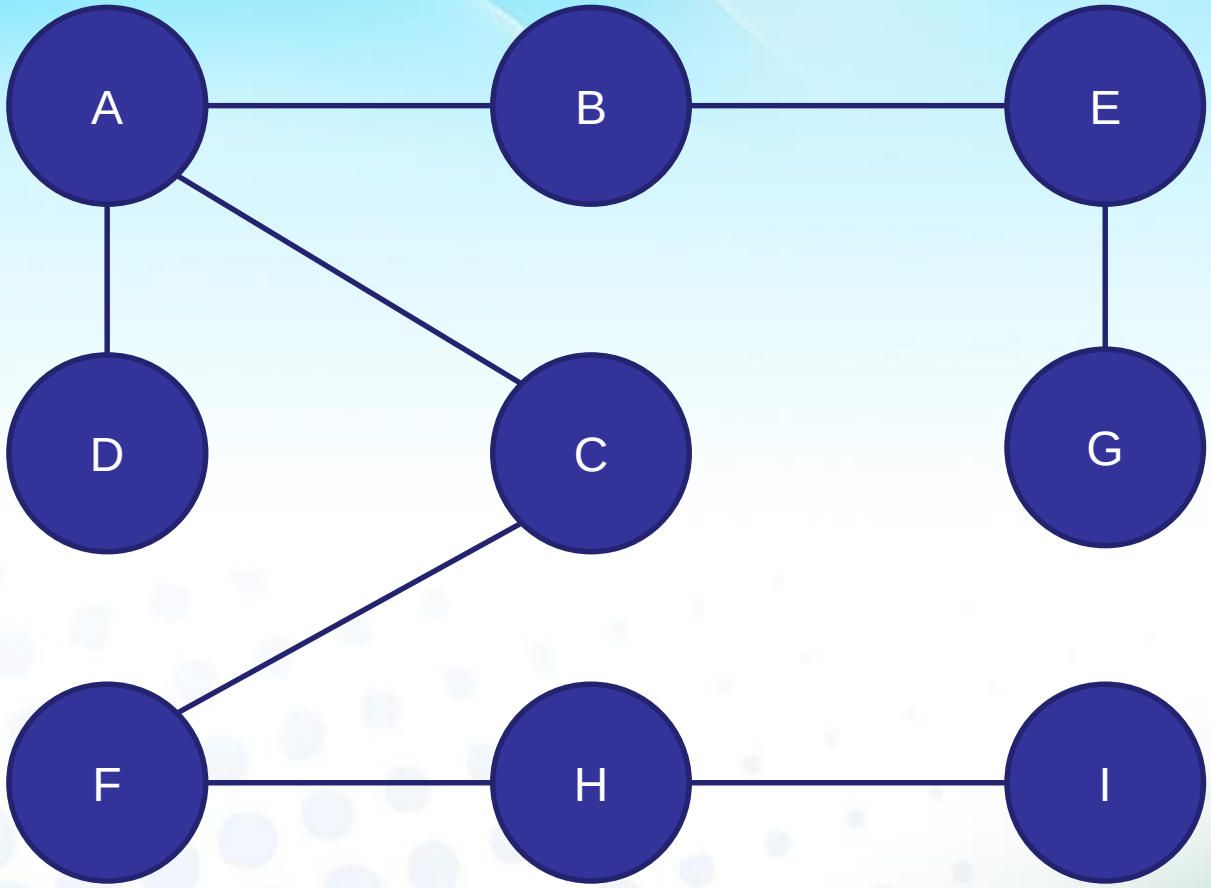
广度优先搜索



广度优先搜索

- 上图给出了一个广度优先搜索的实例，从顶点A出发做广度度优先搜索，可以逐层遍历该图的所有顶点
- 与深度优先搜索类似，在广度优先搜索的过程中，所有访问过的顶点和经过的边构成了一棵树
- 这棵树称之为图的广度优先生成树（BFS tree），同样有很高的应用价值

广度优先生成树



广度优先生成树

- 广度优先搜索一般使用队列，以记忆正在访问的这一层和上一层的结点，以便于向下一层的结点进行访问
- 与深度优先搜索的过程相比，广度优先搜索的过程中多了一项最短距离d的计算，而这通常是我们进行广度优先搜索能够获得的最重要的信息，因此特别在伪代码中列出其过程

广度优先搜索的伪代码

```
BFS(G, s)
for each vertex u in V[G] do
        visited[u] ← false
visited[s] ← true
d[s] ← 0
EnQueue(Queue, s)
while (Queue is not empty) do
        u ← DeQueue(Queue)
        for each v in the adjacency list of u do
                if (not visited[v])  then
                        visited[v] ← true
                        d[v] ← d[u] + 1
                        EnQueue(Queue, v)
```

遍历算法总结

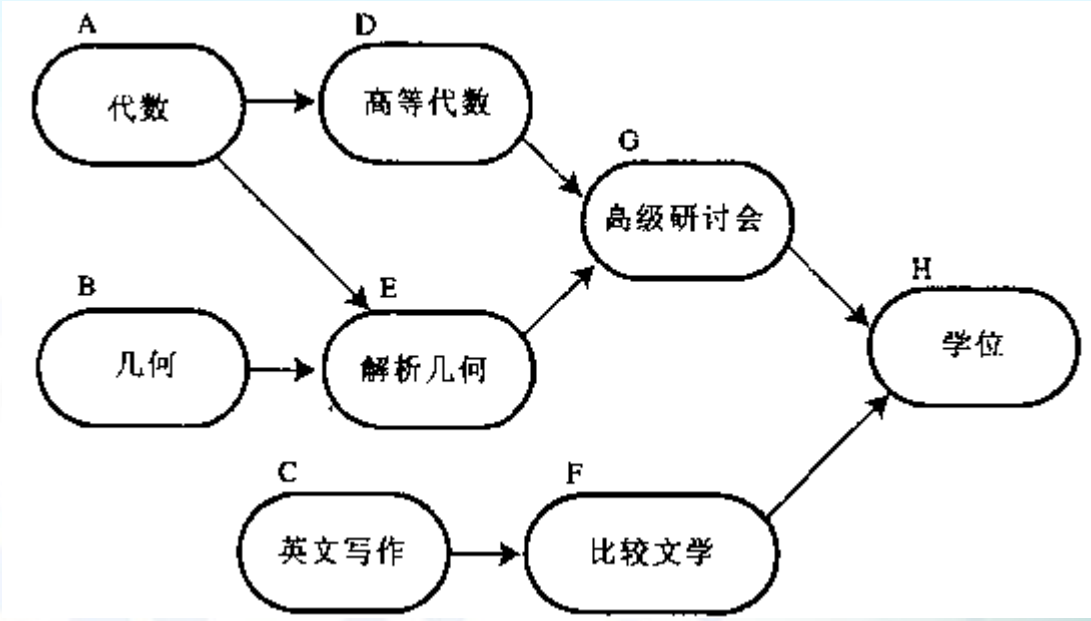
- 对于我们的两种遍历算法，每个顶点均至多只入栈出栈一次或是入队出队一次，每次访问其关联的所有边，因此总共的访问次数不会超过(2|*V*|+|*E*|)的级别，是十分高效的
- 一般来说，广度优先搜索通常用于从某个起始点开始寻找最短距离，深度优先搜索通常作为另一个算法中的子程序

拓扑排序

- 拓扑排序（topological sort）指的是从有向图$G=(V, E)$中得到一个顶点的线性序列，满足如果G包含边(u, v)，则在该序列中，u就出现在v的前面
- 一个图的拓扑排序可以看成是图中所有顶点沿水平线排列而成的一个序列，使得所有的有向边均从左到右
- 因此拓扑排序不同于我们学习过的排序

拓扑排序

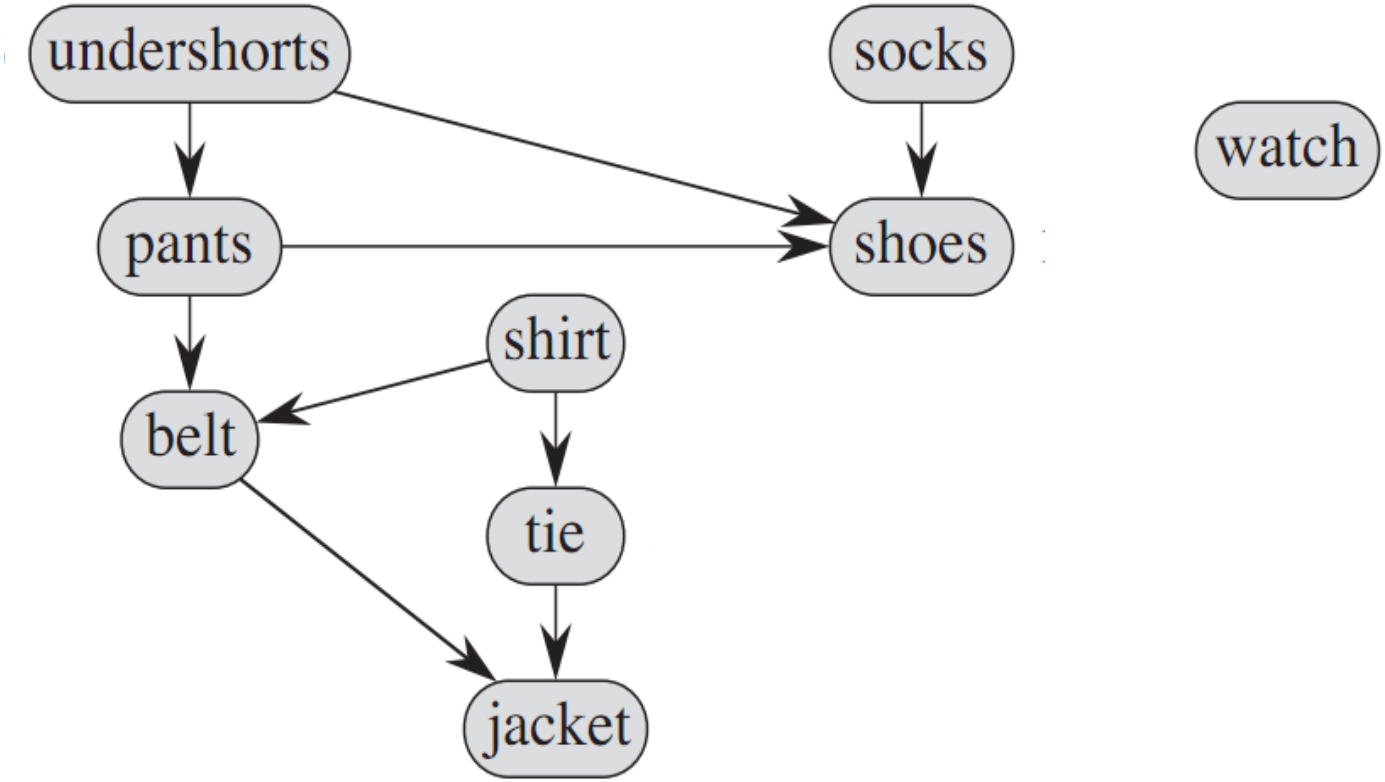
- 在很多应用中，有向图用于说明事件发生的先后顺序，比如下图的课程先决依赖关系，拓扑排序的结果就是可行的修读顺序



拓扑排序

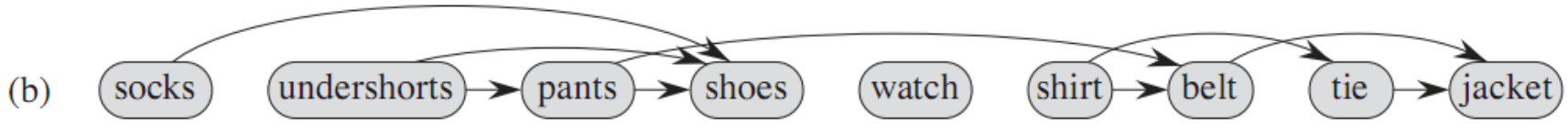
- 对于早晨穿衣的过程，一个人必须先穿好某些衣服，才能穿其他衣服（如先穿袜子后穿鞋），其它的衣服则可以按任意次序穿戴（如袜子和裤子）
- 在图中，我们用有向边(u, v)表示衣服u必须先于衣服v穿戴
- 注意到如果图有回路的话，那么会形成循环依赖，也就不可能有合适的穿衣顺序了

拓扑排序



拓扑排序

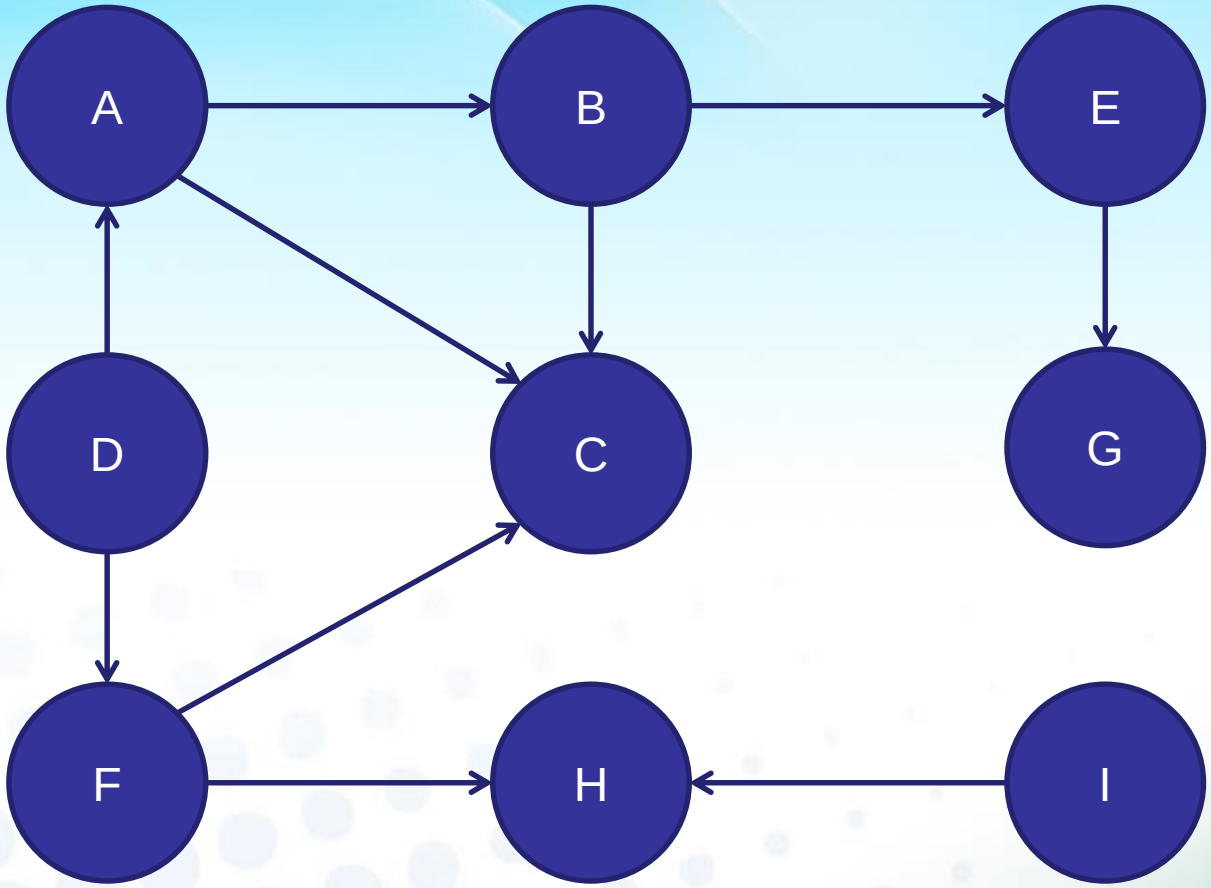
- 下面为对上图拓扑排序后的结果，顶点沿水平线方向形成一个顶点序列，使得图中所有有向边均从左至右，从而这就是一个合理的穿衣的顺序
- 需要注意的是，这样的顺序并不是唯一的



拓扑排序的算法

- 拓扑排序有一个非常简单的算法，可以概述如下：
  - 从有向图中选择一个入度为0的顶点输出，因为这意味对该顶点没有限制要求
  - 从有向图中删去该顶点,并且删去从该顶点发出的全部有向边，即在其完成后去除来自于该顶点对于其它顶点的限制要求
  - 重复上述两步,直到所有的顶点都已处理
- 大家可以使用上面的图例自行验证

拓扑排序



拓扑排序的伪代码

```
TopSort(G)
for each vertex u in V[G] do
        if (indeg(u)=0)
                then EnQueue(Queue, u)
while (Queue is not empty) do
        u ← DeQueue(Queue)
        for each v in the adjacency list of u do
                indeg(v) ← indeg(v)  - 1
                if (indeg(v)=0)
                        then EnQueue(Queue, v)
```

进阶内容

- 深度优先搜索和广度优先搜索还有许多有趣的性质，如括号定理，白色路径定理，反向边定理等等
- 其中，利用白色路径定理，可以导出拓扑排序的另一种算法——通过计算深度优先搜索的结点探查结束时间来进行拓扑排序

Thank you

Q&A