数据结构与算法B

12-图

目录

12.1 图的基本概念

数据的逻辑结构

- 线性结构：唯一前驱，唯一后继，线性关系
- 树结构：唯一前驱，多个后继，层次关系
- **图结构：多个前驱、多个后继，网状关系**

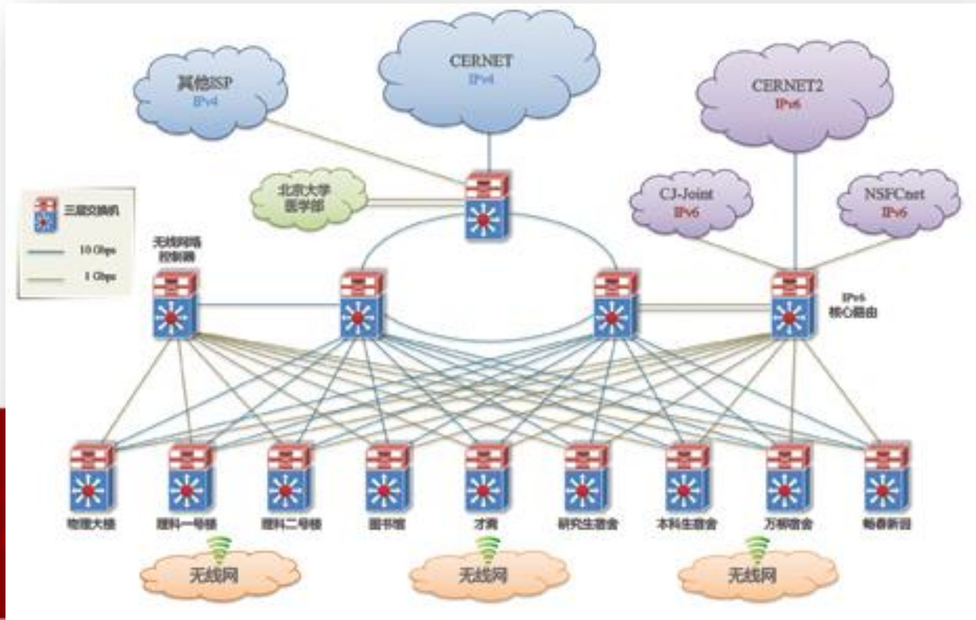
	集合	线性结构	树形结构	图状或网状结构
特征	元素间为松散的关系	元素间为一对一关系	元素间为一对多关系	元素间为多对多关系
示例		a→b→c→d		

生活中的图结构

- 北京公共交通
  - 北京地铁共有18条运营线路，不重复计算换乘车站则为268座车站，总长约527千米。
  - 北京公交系统有1020条运营线路，公交站点近2000个。

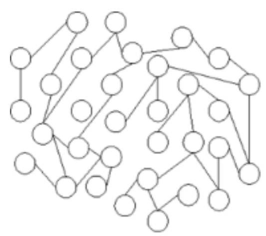
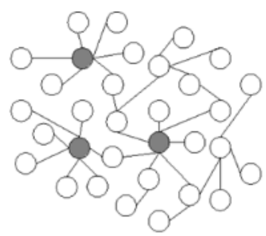
生活中的图结构

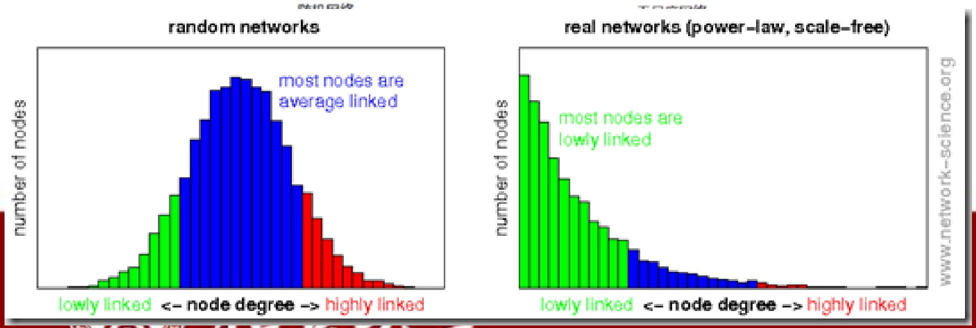
- 互联网
  - 北京大学校园网目前已经具有近10万信息点
  - 通过层层的交换机、路由器连接在一起，路由器之间又相互连接
  - 全球互联网中ipv4的近40亿地址已接近枯竭
    - 一张几十亿个信息点的巨型网络
  - 提供内容的Web站点已突破10亿个
    - 由超链接相互连接的网页更是不计其数，Google每天处理的数据量约10PB



生活中的图结构

- 社交网络：六度分隔理论
  - 世界上任何两个人之间通过最多6个人即可建立联系
  - 在社会中有20%擅长交往的人，建立了80%的连接
- 无尺度网络：只有少数结点与大量结点连接
  - 区别于随机网络
  - 现实中的复杂网络多属于无尺度网络

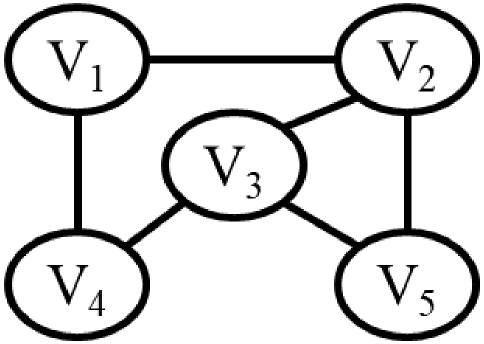


图的基本概念

- 图的逻辑结构：G = (V, E)
  - 图由顶点集合与边集合组成
  - V为有穷的顶点集合
  - E为边集合，是顶点的偶对（边的始点，边的终点）集合
  - 用 |V| 表示顶点的总数，|E| 表示边的总数
- 不同特点的图：
  - 稀疏图(sparse graph) / 稠密图(dense graph)：边数相对较少 / 较多
  - 完全图(complete graph)：包含所有可能的边
  - 有向图(directed graph) / 无向图(undirected graph)：顶点对有序 / 无序
  - 带权图(weighted graph)：边上标有权的图
  - 标号图(labeled graph)：各顶点均带有标号的图

无向图

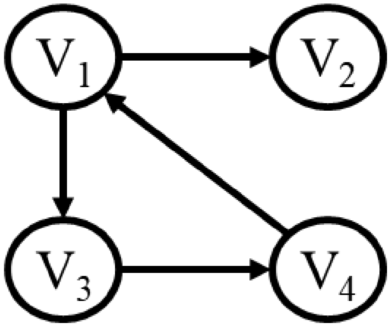
- 无向图：若图中每条边都是无方向的，则称为无向图。
  - 无向图中的边是由两个顶点组成的无序对
  - 无序对用圆括号表示，如(v_i, v_j)；(v_i, v_j)和(v_j, v_i)代表同一条边。
  - v_i和v_j是相邻结点，(v_i, v_j)是与顶点v_i和v_j相关联的边。



无向图

有向图

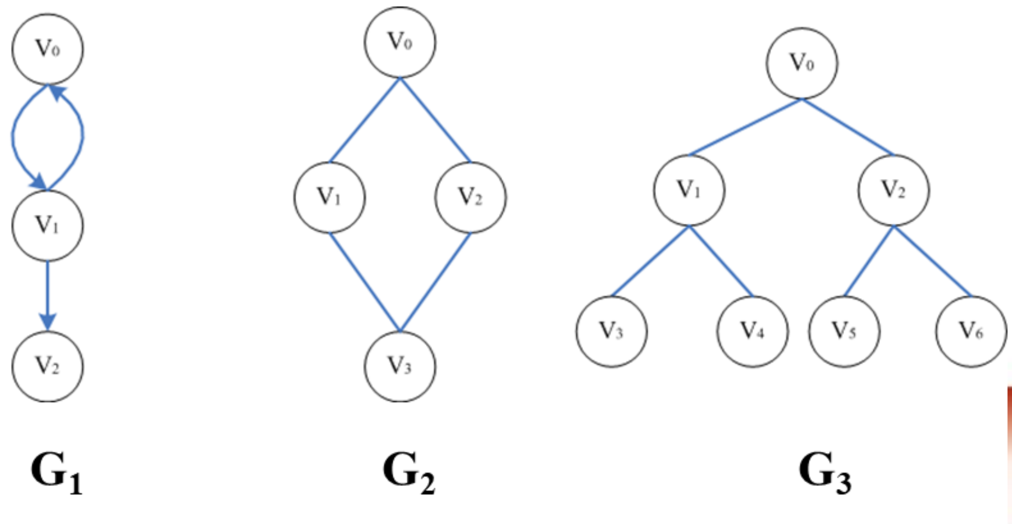
- 有向图：若图中每条边都是有方向的，则称为有向图。
  - 有向图中的边是由两个顶点组成的有序对。
  - 有序对用尖括号表示，如< v_i ,v_j >。v_i是边的始点，v_j是边的终点。
  - < v_j ,v_i >和< v_i ,v_j >表示不同的边。
  - 边< v_i ,v_j >与顶点 v_i , v_j 相关联



有向图

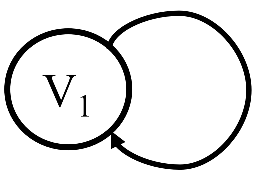
无向图与有向图：示例

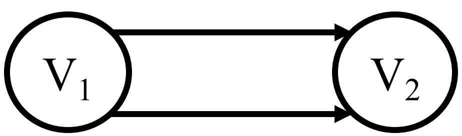
- G_1为有向图
  - $G_1=(V_1, E_1)$、$V_1=\{v_0, v_1, v_2\}$、$E_1=\{<v_0, v_1>, <v_1, v_0>, <v_1, v_2>\}$
- G_2和G_3都是无向图
  - $G_2=(V_2, E_2)$、$V_2=\{v_0, v_1, v_2, v_3\}$, $E_2=\{(v_0, v_1), (v_0, v_2), (v_1, v_3), (v_2, v_3)\}$
  - $G_3=(V_3, E_3)$、$V_3=\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$、$E_3=\{(v_0, v_1), (v_0, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_5), (v_2, v_6)\}$
- G_1, G_2, G_3都是标号图



简单图

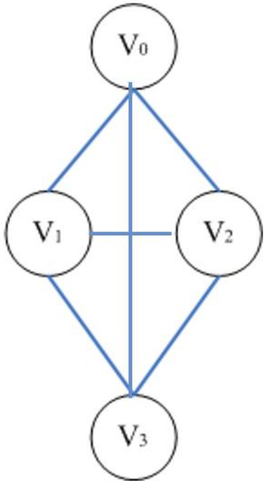
- 自环：一条边的两个端点是同一个顶点
- 重边：无向图中两个顶点之间有不止一条边，或有向图中两个顶点之间有不止一条同方向的边
- 简单图：不包含自环或重边的图
- 约定：除非特别说明，后续讨论的图都是简单图





完全图

- 完全图：图中的任意两个顶点之间都存在边
  - 对于有向图：任意两个顶点之间存在任意方向的边
- 记 n = |V|, e = |E|
- 若G是有向图，则$0 \le e \le n(n-1)$
  - 有向完全图具有n(n-1)条边
- 若G是无向图，则$0 \le e \le n(n-1)/2$
  - 无向完全图具有n(n-1)/2条边
- 在顶点个数相同的图中，完全图具有最多的边
  - 右图就是一个具有4个顶点的无向完全图，边数为:4*(4-1)/2=6



顶点的度数

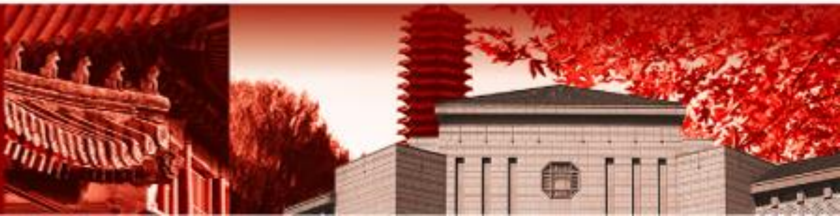
- 在无向图中：
  - 与顶点v相关联的边数称为顶点v的度数，记为D(v)
- 在有向图中：
  - 入度：以v为终点的边的数目称为v的入度，记为ID(v)
  - 出度：以v为始点的边的数目称为v的出度，记为OD(v)
  - v的度数为其入度和出度之和，即D(v)=ID(v)+OD(v)
- 回忆对比：树结构中的度数
  - 结点的度数定义为子女的个数
  - 树的度数定义为所有结点度数的最大值
    - 如果将树视为无向图，度数还应该考虑连接父节点的边

顶点的度数

- 若图G有n个顶点，e条边，则有

$$e = \frac{1}{2}\sum_{i=1}^{n} D(v_i)$$

- 每一条边都参与计算了其两个顶点的度数
- 有向图与无向图均满足这一关系

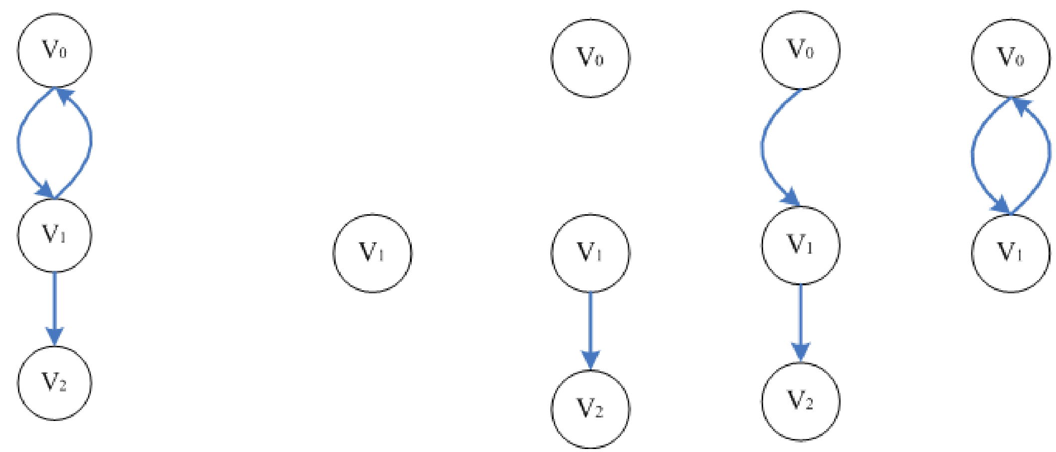


相邻结点

- 在无向图中：
  - 若存在边(x, y)，则顶点x, y互为相邻结点，或者说x, y相邻。
- 在有向图中：
  - 若存在边<x, y>，即存在边由x指向y，则顶点y是顶点x的相邻结点。
- 相邻结点，也称为邻居、邻居结点、邻点等。

子图

- 定义：设有图G=(V，E)和G’=(V’，E’)，如果V’是V的子集，E’是E的子集，则称G’是G的子图。
  - 下图给出了有向图G_1的若干子图。



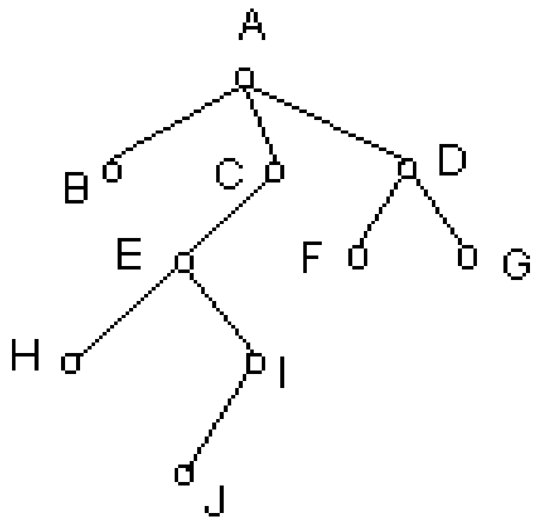
(G_1)

路径

- 定义：对于无向图G=(V, E)，若存在顶点序列$v_{i0}, v_{i1}, \ldots, v_{in}$，使得$(v_{i0}, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{in-1}, v_{in})$都在E中，则称从顶点$v_{i0}$到$v_{in}$存在一条路径。
  - 对于有向图G：只需要将条件修改为：$<v_{i0}, v_{i1}>, < v_{i1}, v_{i2}>, \ldots, <v_{in-1}, v_{in}>$都在E中
- 路径长度：路径上的边数。
- 简单路径：路径上的顶点除v_{i0}和v_{in}可以相同外，任意两个顶点都不相同。
- 回路或环：起点和终点相同的简单路径。

路径

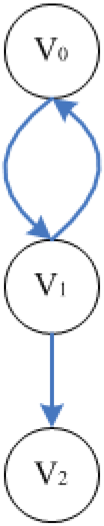
- 回忆对比：树中的路径
  - 对树上任意两个节点V_i，V_j，必然存在唯一的不重复的结点序列$\{V_i, P_1, P_2, \ldots, P_m, V_j\}$，使得$V_i$和$P_1$之间有边，$P_m$和$V_j$之间有边，$P_k$和$P_{k+1}$之间有边，这个结点序列就被称作为$V_i$到$V_j$的路径。
  - 路径上边的数量被称作是路径长度。



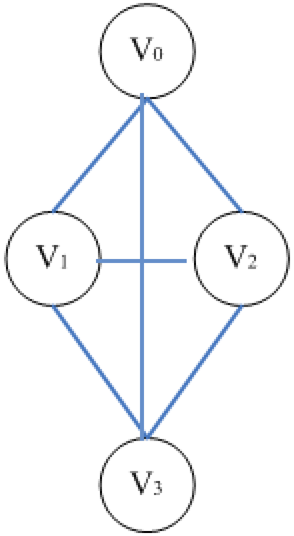
- A是其它各结点的祖先；C是E，H，I，J的祖先
- A，C，E，I，J是从A到J的一条路径，其长度为4

路径：示例

- 在图G_1中
  - 顶点序列v_0,v_1,v_0是一长度为2 的有向环。
- 在图G_2中
  - 顶点序列v_0,v_1,v_2,v_3是一从顶点v_0到v_3的长度为3的路径；
  - 顶点序列v_0,v_1,v_3,v_0,v_2是一从顶点v_0到v_2的长度为4的路径，但不是简单路径；
  - 顶点序列v_0,v_1,v_3,v_0是一长度为3的环。



(G_1)



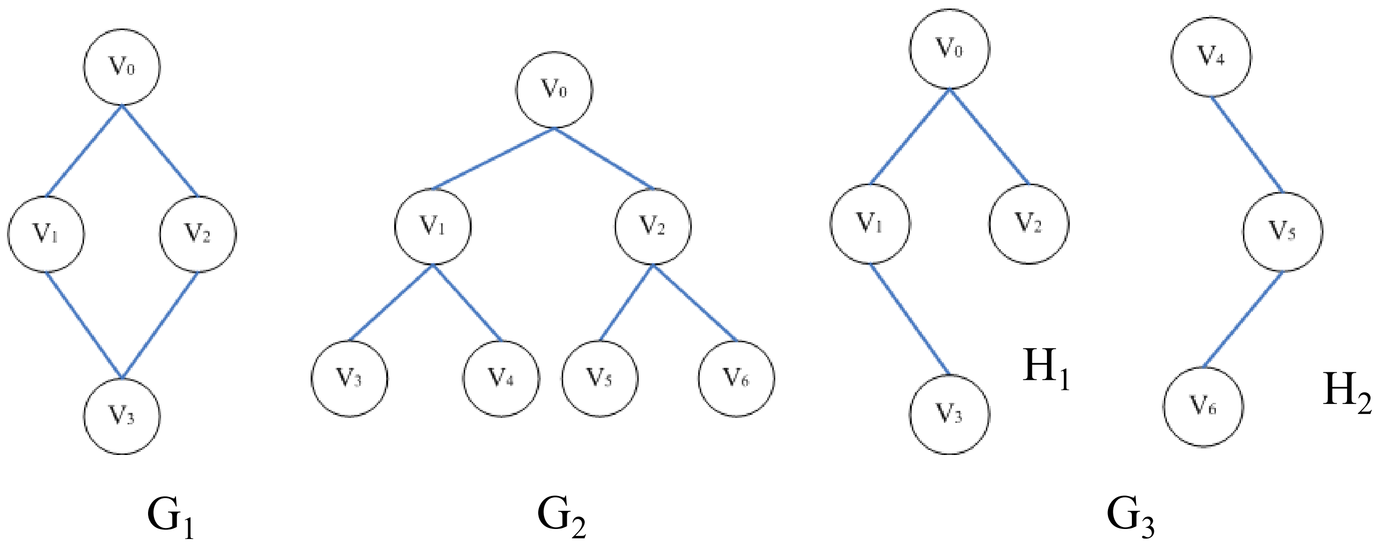
(G_2)

无向图的连通性

- 无向图的连通性
  - 连通：无向图G=(V，E)中，若从v_i到v_j有一条路径(从v_j到v_i也一定有一条路径)，则称v_i和v_j是连通的。
  - 连通图：若V(G)中任意两个不同的顶点v_i和v_j都是连通的(即有路径)，则称G为连通图。
  - 连通分量：无向图G的极大连通子图G’（极大：即任意增加G中结点和边到G’所得到的子图都不再连通），称为G的连通分量。
- 连通图只有一个连通分量，就是其自身；
- 非连通的无向图有多个连通分量。

无向图的连通性：示例

- 无向图连通的例子：
  - G_1和G_2都是连通图
  - G_3是非连通图，它有两个连通分量H_1和H_2

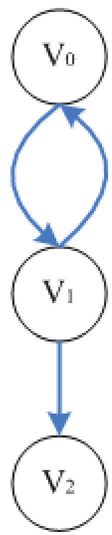


有向图的连通性

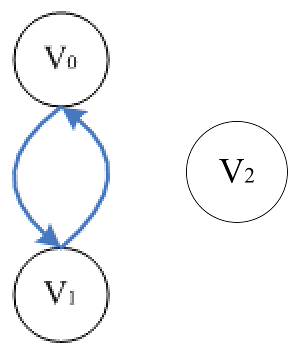
- 有向图的连通性
  - 强连通图：有向图G=(V，E)中，若V中任意两个不同的顶点v_i和v_j都存在从v_i到v_j以及从v_j和v_i的路径，则称图G是强连通图。
  - 强连通分量：有向图G的极大强连通子图称为图G的强连通分量。
  - 弱连通图：将有向图G=(V, E)中的所有有向边替换为无向边，得到的无向图如果是连通图，则称有向图G是弱连通图。
    - 即在原有向图中，对于V中任意两个不同的顶点v_i和v_j，存在从v_i到v_j或者从v_j和v_i的路径
  - 弱连通分量：有向图的G的极大弱连通子图称为图G的弱连通分量
- 强连通图只有一个强连通分量，就是其自身；
- 非强连通的有向图有多个强连通分量。

有向图的连通性：示例

- 有向图连通的例子：
  – 下图展示了一个弱连通、但非强连通图G，以及它的两个强连通分量



G



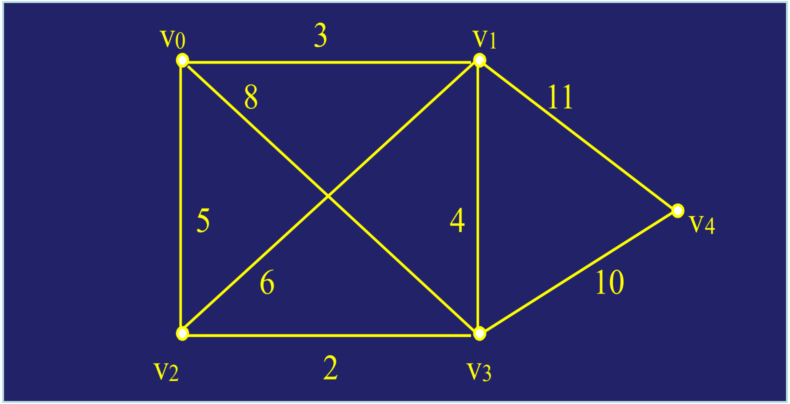
图的根

- 定义：有向图中，若存在一顶点v，从该顶点到图中其它所有顶点都存在路径，则称此有向图为有根图，v称为图的根
  - 连通的无向图中这一概念是平凡的

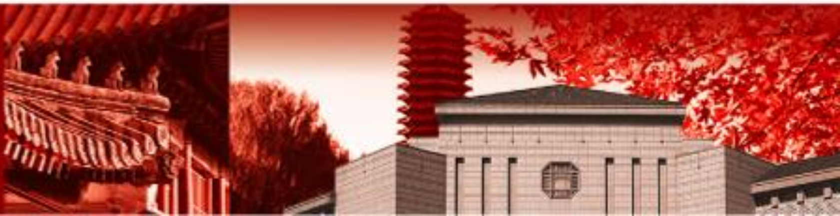
图中的根可能
不唯一！！

带权图

- 若给图的每条边都赋上一个权值，则称该图为带权图
  - 可以是无向图或者有向图
  - 通常这一权重具有实际意义，例如顶点间的距离、通信的花费、或者边所表示的关系的强弱等
- 连通的带权图也称为网络
- 下图展示了一个带权图（网络）



12.2 图的存储方式



存储结构

- 回忆我们已经学习过的存储结构
- 顺序方法：利用元素地址的连续性来维持逻辑结构
  - 顺序表：后继即为下标+1的元素
  - 二叉树的顺序存储：下标为i的元素的左子、右子下标分别为2i, 2i+1
- 链式方法：利用引用（指针）来维持逻辑结构
  - 例如：链表、二叉树的链式实现等
- 图逻辑结构中，每个元素可以有任意多个相邻结点，且需要区分边是否有向，因而图的存储方法也更复杂。
- 应根据具体的应用和施加的操作选择不同的存储表示法
  - 邻接矩阵表示法
  - 邻接表表示法

邻接矩阵表示法

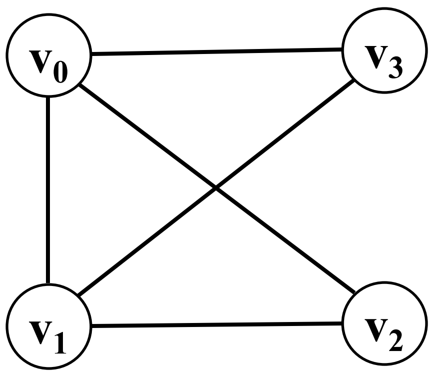
- 设G=(V, E)为具有n个顶点的图，其邻接矩阵(Adjacency matrix) A 为如下定义的 n 阶方阵

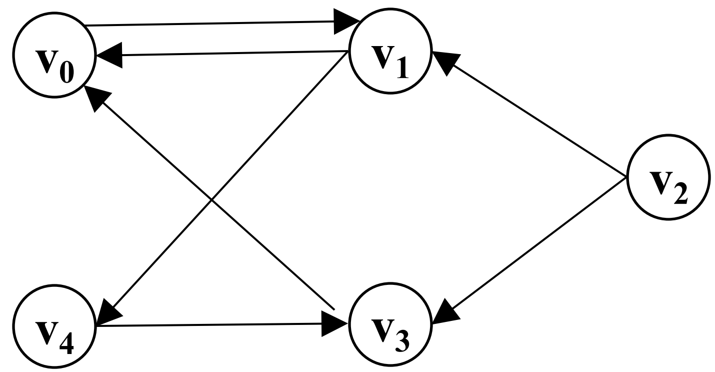
$$A[i,j]=\begin{cases}1, & \text{如果}(v_i,v_j)\text{或} < v_i,v_j > \text{是图}G\text{的边}\\ 0, & \text{如果}(v_i,v_j)\text{或} < v_i,v_j > \text{不是图}G\text{的边}\end{cases}$$

- 存储图的邻接矩阵，就维护了所有顶点以及顶点之间的相邻关系
  - 顶点信息：依次标号为0 ~ n-1
  - 关系信息：每条边都对应到邻接矩阵中的一个非零元素
- 对于不带权的无向图，邻接矩阵A是对称的；对于带权图或者有向图，A一般非对称

邻接矩阵表示法：示例

- 无向图(左)和有向图(右)的邻接矩阵分别为A_1和A_2
- A_1是对称矩阵，A_2是非对称矩阵





$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

邻接矩阵表示法：带权图

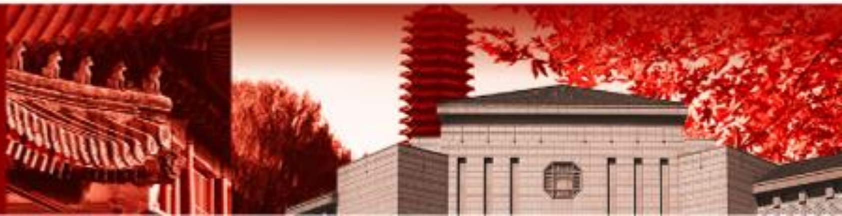
- 如果G是带权的图，w_{ij}是边(v_i,v_j)或$<v_i,v_j>$的权，则其邻接矩阵定义为：

$$A[i,j]=\begin{cases} w_{ij}, & \text{如果}(v_i,v_j)\text{或} < v_i,v_j > \text{是图}G\text{的边} \\ \infty \text{ 或 } 0, & \text{如果}(v_i,v_j)\text{或} < v_i,v_j > \text{不是图}G\text{的边} \end{cases}$$

- 同样地，无向图的带权邻接矩阵是对称矩阵，有向图的带权邻接矩阵通常是非对称逆矩阵。
- 如果边(v_i,v_j)或$<v_i,v_j>$不存在，权重通常设置为无穷或0：
  - 如果权重表示结点间的距离等含义，则应该设置为无穷
  - 如果权重表示结点之间的流量、结点之间关系的强弱等含义，则应该设置为0。
  - 实际应用中也可以设定为其他约定好的特殊值，如 -1

邻接矩阵表示法：实现

- 最直观实现方法：使用一个嵌套列表 L 存储邻接矩阵 A
  - 每一个子列表存储邻接矩阵的一行
  - L[i][j] = A[i, j]
- 计算结点的度数：
  - 无向图中第 i 个结点的度数 $D(v_i)$：第 i 行/列中非零元素的个数
  - 有向图中第 i 个结点的出度 OD(vi)：第 i 行中非零元素的个数
  - 有向图中第 i 个结点的出度 ID(vi)：第 i 列中非零元素的个数
- 寻找结点的相邻结点：
  - 无向图中第 i 个结点的相邻结点：第 i 行中非零元素对应的下标
  - 有向图中第 i 个结点的相邻节点：第 i 行中非零元素对应的下标

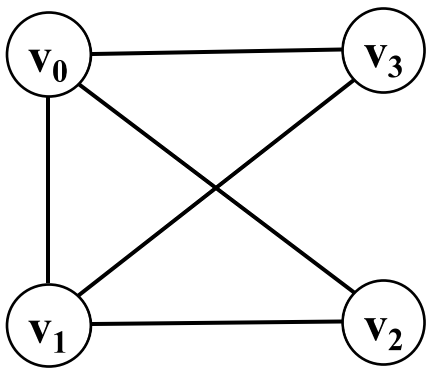


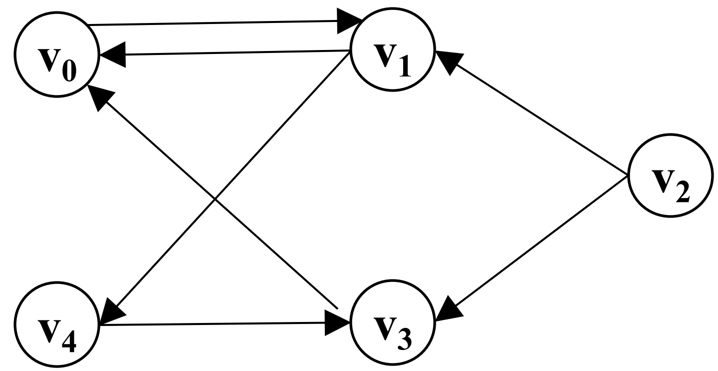
邻接表表示法

- 为了记录图中所有的边，可以为每个顶点 x 设置一个线性表，记录 x 的所有相邻结点。这种图的表示方式称为邻接表。
  - 该线性表是不定长的，x 有几个相邻结点，对应线性表中就有几个元素
  - 线性表中的一个元素记录了一个相邻结点 y，表示存在边 (x, y) 或 <x, y>
  - 对于有权图，或者边附带有其他信息，也可以存储在该线性表中

邻接表表示法：示例

- 无向图(左)和有向图(右)的邻接表分别为 G_1和 G_2





G_1=[[1, 2, 3],
[0, 2, 3],
[0, 1],
[0, 1]]

G_2=[[1],
[0, 4],
[1, 3],
[0],
[3]]

邻接表表示法：实现

- 邻接表表示法同样可以用嵌套列表实现
- 对于嵌套列表G，G[i] 就是一个一维列表，称为G的一行
  - G[i] 中存放结点 V_i 的所有相邻结点
  - 对于带权图，G[i]中的元素可以是元组，即（相邻结点，对应边权）
  - 具体实现时，如果结点没有额外信息，G[i]中只存放结点标号即可；否则，也可以首先将结点实现为类，G[i] 中存放结点对象的引用
- Python中的列表本身就是不定长的，各行之间的长度也通常不同，*G*本身通常不构成一个矩阵
- 邻接表表示法就是基于相邻结点进行存储的。该表示法下，计算结点的度数、寻找结点的相邻结点等操作更加简单。

邻接矩阵表示法 vs 邻接表表示法

- 下面分别从不同的角度，比较这两种表示法的优劣
  - 用 n 表示结点数量，e 表示边数量
- 1. 构建一个图的复杂度
  - 邻接矩阵表示法：时间、空间复杂度都是 $O(n^2)$ 的。这意味着对于任何问题，如果采用这种存储方式，时间复杂度都至少是 $O(n^2)$
  - 邻接表表示法：时间、空间复杂度都是 $O(n+e)$ 的。尽管 e 理论上能够达到 n^2 的量级，但实际大多数情况下远小于 n^2
  - 思考：$O(n+e)$ 能够写成 $O(e)$ 吗？
- 2. 对任意的两个结点 V_i, V_j，查询它们之间边的情况
  - 邻接矩阵表示法：$O(1)$ 的复杂度就可以访问 A[i][j] 元素
  - 邻接表表示法：需要遍历 G[i] 中的元素，复杂度为 $O(n)$

邻接矩阵表示法 vs 邻接表表示法

- 3. 对于有向图，求顶点 V_i 的入度
  - 邻接矩阵表示法：查询邻接矩阵A的第 i 列即可，复杂度为O(n)
  - 邻接表表示法：遍历每一行查找 V_i，复杂度为O(n+e)
- 4. 查找顶点 V_i 的相邻结点，以及边的权值
  - 邻接矩阵表示法：查询邻接矩阵的第 i 行即可
  - 邻接表表示法：查询邻接表的第 i 行即可
  - 理论上二者的复杂度都是 O(n)。但是，邻接表只需要存储图中存在边，而邻接矩阵中还包含若干个 0 或无穷。相比之下，前者的信息更加紧凑，查询也会更快

邻接矩阵表示法 vs 邻接表表示法

- 5. 存储稀疏图的效率
  - 邻接矩阵表示法：时间空间复杂度与图是否稀疏无关，$O(n^2)$
  - 邻接表表示法：每条边只需要存储一次（有向图）或两次（无向图），复杂度为 $O(n+e)$。如果 e 远小于 n^2，效率能够大大提高
- 现实中的图，绝大多数是稀疏图，而且找出相邻结点以及相连的边，基本上是最常用的操作。因此，邻接表表示法的应用比邻接矩阵表示法更加普遍。
- 如果要表示稠密图，或对图的操作主要是查询结点间是否存在边关系，使用邻接矩阵表示法就比较合适。

12.3 图的搜索与遍历

图的搜索与遍历

- 定义：
  – 图的搜索：从图中给定的一个起始顶点出发，寻找一条到目标顶点的路径，称为图的搜索。
  – 图的遍历：从图中某一顶点出发，按照某种方式系统地访问图中所有顶点，使得每一个顶点被访问且仅被访问一次，称为图的遍历，也称为图的周游。
- 搜索有两种基本策略：深度优先搜索（Depth First Search, DFS）以及广度优先搜索（Breadth First Search, BFS）
- 这两种策略同样可以用于图的遍历，即深度优先遍历与广度优先遍历。因此，图的搜索与遍历是联系紧密的两种操作

图的搜索问题

- 对于许多问题的本质，都可以抽象为在图上的两个顶点之间寻找路径
  - 可以从问题中抽象出一系列状态，对应图中的顶点集合
  - 状态之间的可以转换，对应图中的边集合
  - 问题本身就是求解由一个状态迁移到另一个状态的过程
  - 这一表示过程就是问题的建模

图的搜索问题：示例

- 例1：走迷宫问题，右图表示了一个迷宫
  - . 表示可以走的空地，# 表示墙
  - 入口和出口分别为左上角和右下角
  - 状态即为行进中的位置
  - 如果两个位置一步可达，就在两个状态之间连一条无向边
  - 寻找由左上顶点到右下顶点的一条路径

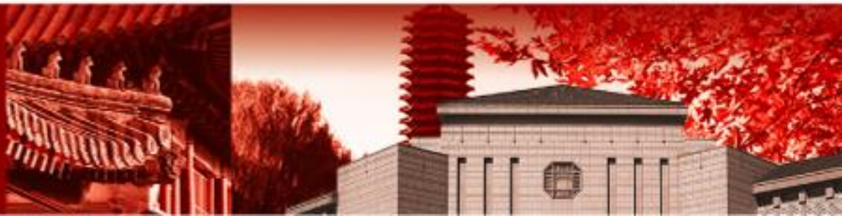
```
.#######
...#....
.#.#.##.
.#....#.
.####.#.
........
```

图的搜索问题：示例

- 例2：解魔方问题
  - 可以采用合适的方法，表示出所有可能出现的魔方局面（状态）
  - 初始状态为某个随机打乱的状态，目标状态为六面对齐的状态
  - 如果状态A经过一次转动（90度）就能到达状态B，就在两个状态间连一条无向边
  - 在这一无向图中，寻找一个从给定的初始顶点（打乱状态）到目标顶点（完成状态）的路径
  - 实际上，这是一个巨大的图，总顶点数是一个天文数字

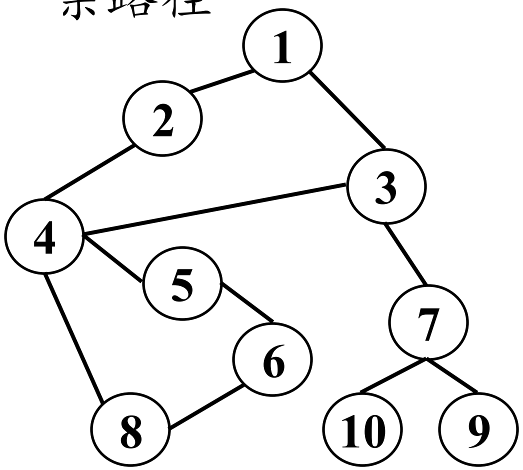
图的搜索问题：示例

- 例3：24点问题，给定4个数，利用加减乘除计算出 24
  - 状态定义为当前的数字集合（集合元素允许重复）
  - 目标状态的集合只有一个元素，即 24
  - 从状态A中取出两个数字，计算后得到状态B，就连接一条由 A 到 B 的有向边
- 尽管这类可以建模为图上的搜索问题，但有时图的完整规模过大，对于有些问题甚至是无穷的。
- 实际上，并不需要用邻接表或邻接矩阵先表示出全部状态；
- 只需要做到，能够找出一个结点的相邻结点，就可以在图上进行搜索。而这可以依据具体的问题场景动态计算。



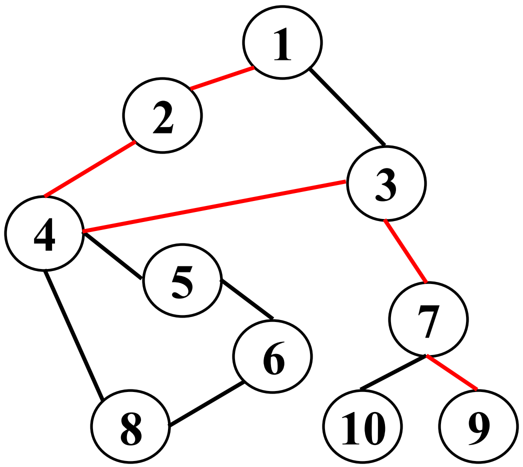
深度优先搜索

- 下面给出了一个图的搜索问题示例：在无向图中，寻找由结点1到结点8的路径
- 最简单的策略之一（DFS）：能向前走就向前走，无法前进时就回溯到之前的位置，尝试其他的路径
  – 在有多条路径可供选择时，可以任意/随机选择一条路径
  – 如果运气较好：1 – 2 – 4 – 8



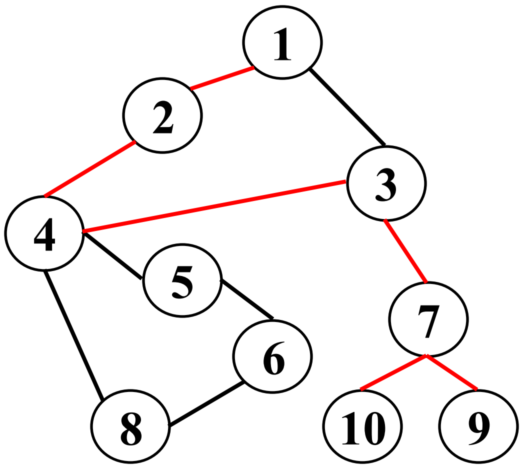
深度优先搜索

- 需要回溯的情况
  - 如果运气不那么好：1 – 2 – 4 – 3 – 7 – 9，无路可走时失败



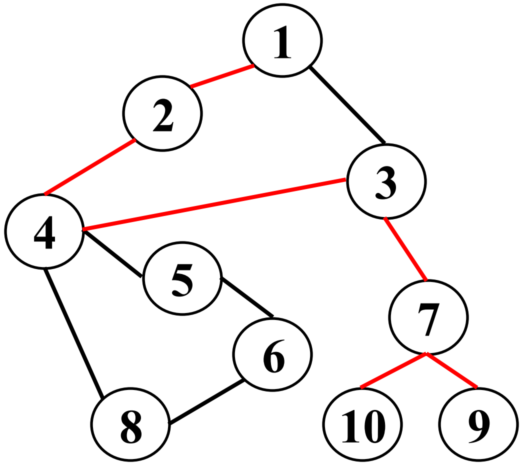
深度优先搜索

- 需要回溯的情况
  - 如果运气不那么好：1 – 2 – 4 – 3 – 7 – 9，无路可走时失败
  - 回溯至7，尝试 7 连接的其他路径：7 – 10，也失败了



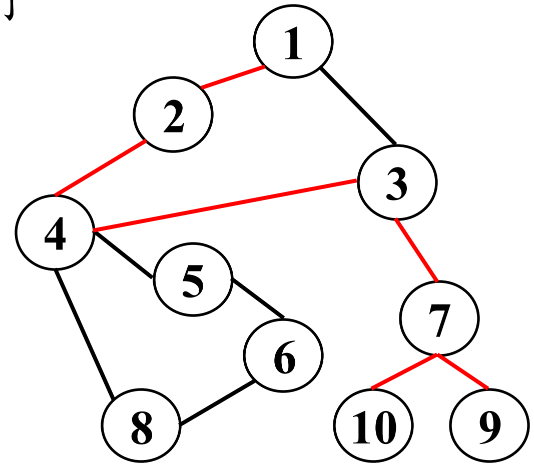
深度优先搜索

- 需要回溯的情况
  - 如果运气不那么好：1 – 2 – 4 – 3 – 7 – 9，无路可走时失败
  - 回溯至7，尝试 7 连接的其他路径：7 – 10，也失败了
  - 回溯至3，尝试 3 连接的其他路径。



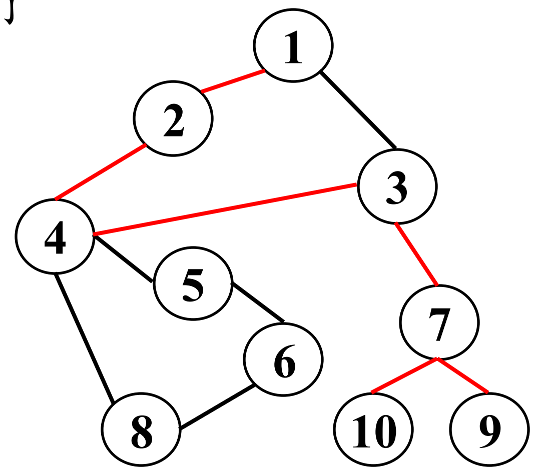
深度优先搜索

- 需要回溯的情况
  - 如果运气不那么好：1 – 2 – 4 – 3 – 7 – 9，无路可走时失败
  - 回溯至7，尝试 7 连接的其他路径：7 – 10，也失败了
  - 回溯至3，尝试 3 连接的其他路径。
    - 另一条边是：3 – 1
    - 不过，显然不应该由 3 走到 1，因为 1 已经被走过了



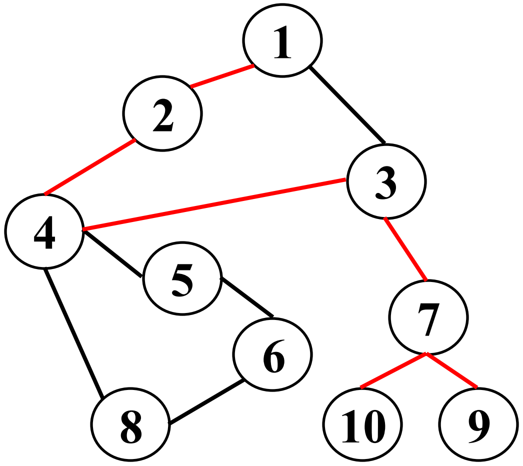
深度优先搜索

- 需要回溯的情况
  - 如果运气不那么好：1 – 2 – 4 – 3 – 7 – 9，无路可走时失败
  - 回溯至7，尝试 7 连接的其他路径：7 – 10，也失败了
  - 回溯至3，尝试 3 连接的其他路径。
    - 另一条边是：3 – 1
    - 不过，显然不应该由 3 走到 1，因为 1 已经被走过了
    - 因此，在搜索的过程中，需要记录已经走过的结点
    - 走过的结点不可以重复走到，这个操作称为判重
    - 因此 3 可能的所有路径都失败了，回溯至 4



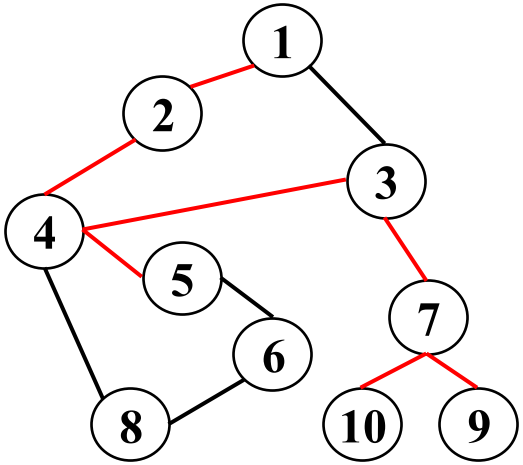
深度优先搜索

- 需要回溯的情况
  - 如果运气不那么好：1 – 2 – 4 – 3 – 7 – 9，无路可走时失败
  - 回溯至7，尝试 7 连接的其他路径：7 – 10，也失败了
  - 回溯至3，尝试 3 连接的其他路径。经过判重，也失败了



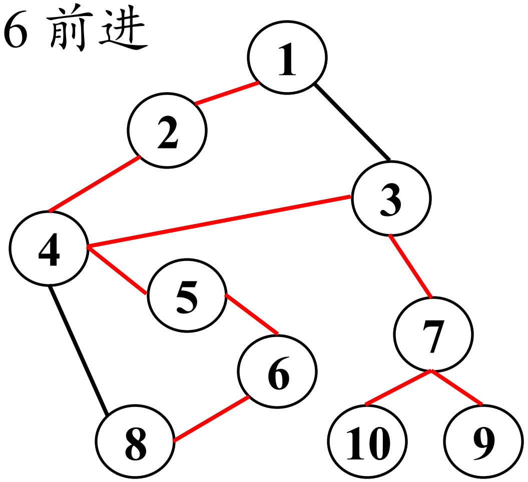
深度优先搜索

- 需要回溯的情况
  - 如果运气不那么好：1 – 2 – 4 – 3 – 7 – 9，无路可走时失败
  - 回溯至7，尝试 7 连接的其他路径：7 – 10，也失败了
  - 回溯至3，尝试 3 连接的其他路径。经过判重，也失败了
  - 回溯至4，假设接下来选择了结点 5



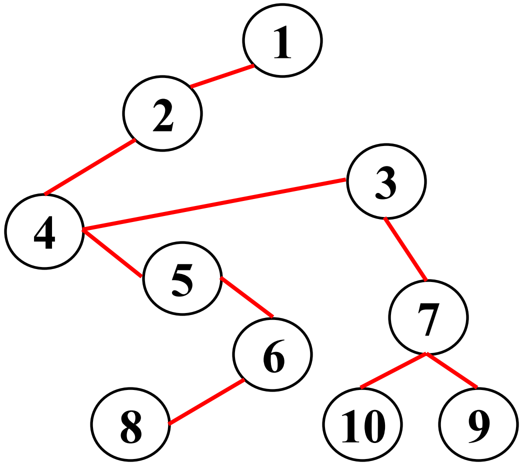
深度优先搜索

- 需要回溯的情况
  - 如果运气不那么好：1 – 2 – 4 – 3 – 7 – 9，无路可走时失败
  - 回溯至7，尝试 7 连接的其他路径：7 – 10，也失败了
  - 回溯至3，尝试 3 连接的其他路径。经过判重，也失败了
  - 回溯至4，假设接下来选择了结点 5
  - 此时经过判重，就不应该向 3 前进，而只能向 6 前进
  - 前进到 6 之后，就自然到达了结点 8



深度优先搜索

- 最终搜索结果：1 – 2 – 4 – 5 – 6 – 8
- 无论选择过程如何，搜索过程中走过的所有结点以及边，都会构成一个树，称为搜索树，如右图
- 搜索的过程，实际上是按深度优先遍历次序访问了搜索树中的所有结点。



深度优先搜索：伪代码

```
# 在图G中寻找一条由start到end的路径 path，不存在则返回 None
def dfs(G, start, end, visited, path):
    将所有顶点设置为未访问
    将 start 结点添加到 path 列表中，并设置为已访问
    if start == end：
        return path
    对于 start 结点的每个邻居 neighbor：
        如果未曾访问 neighbor：
            result = dfs(G, neighbor, end, visited, path)
            如果 result 非空，意味着搜索成功，return result
    搜索失败，return None
```

深度优先搜索：伪代码

```
# 在图 G 中寻找一条由 start 到 end 的路径
# visited 为已访问结点集合，path 为当前路径
def dfs(G, start, end, visited, path):
    将 start 添加到 visited 列表、path 列表中
    if start == end:
        return path
    遍历 start 结点的所有邻居:
        判重，如果未曾访问 neighbor:
            result = dfs(G, neighbor, end, visited, path)      → 递归调用
            如果非空，意味着搜索成功，return result
    搜索失败，return None                                      → 递归出口
```

深度优先搜索

- 对于一些问题（如前述的24点问题），问题建模的图为有向无环图（DAG, Directed Acyclic Graph），由于图中无环，搜索时就不必维护 visited 集合

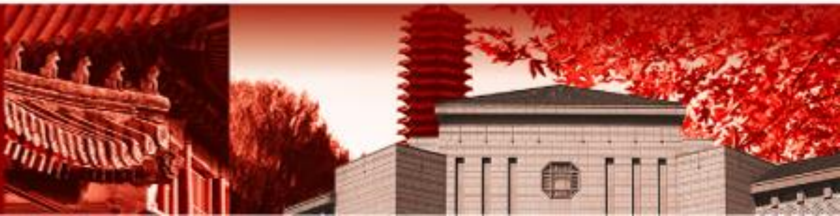
- 深度优先搜索的时间复杂度（假定采用邻接表实现）：
  - 初始化所有结点的时间复杂度为 O(n)
  - 最坏情况下，每个结点都会被访问一次（进入递归），每条边都被检查一次
  - 时间复杂度为 O(n+e)

深度优先搜索：剪枝

- 我们已经看到，实际问题的状态空间可能非常大，因此即使是O(n+e)的复杂度也是不可接受的（魔方问题）
- 搜索过程中，存在一些冗余过程。例如：
  - 根据具体问题性质可以判断出，从某个结点出发无法再到达终点
  - 如果追求路径（或带权路径的权重和）尽可能短，一些代价高昂的路径也可以直接被舍弃掉
- 去除这些冗余的搜索，就是去除了搜索树中的若干分枝。因此，就将这一过程称为剪枝（Pruning）
  - 可行性剪枝：如果可以判断再走下去已经无法到达终点，就可以立即回溯
  - 最优性剪枝：求最小代价/最短路径时，如果当前的代价已经超过当前最优解的代价，就可以立即回溯

深度优先搜索：启发式搜索

- 在图的搜索过程中，不同路径的选择导致不同的搜索树，直接影响搜索的效率。
- 为了尽快地找到可行路径或者最短路径，在路径选择时，可以针对不同的具体问题，使用经验法则或代价估计来指导搜索的方向，从而减小搜索的范围，更快达到目标。
  - 采取的经验法则，可以是经过数学证明的严谨性质，也可以是只基于经验的规则。它们都统称为启发式规则（Heuristics）
  - 在一些问题上，严格证明过的启发式规则可以保证得到最优解。
  - 未经严格分析的启发式规则，也能对搜索效率提供不同程度的提升
  - 启发式搜索也可以视为广义的剪枝策略

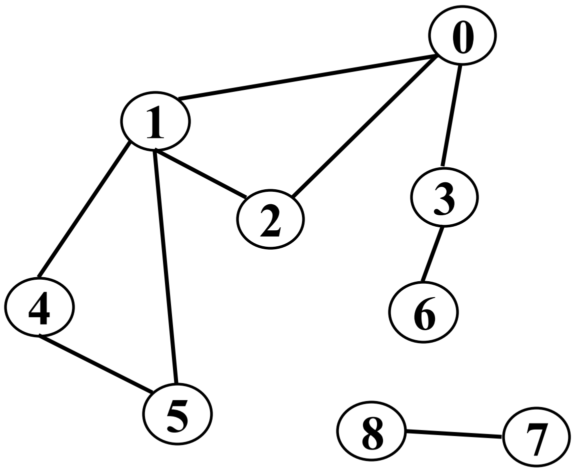


深度优先遍历

- 采用深度优先搜索的策略，遍历图中的所有顶点，称为深度优先遍历。具体步骤如下
  - (1) 在图中任意选择没有走过的顶点，作为遍历的起点
  - (2) 当前顶点如果有未走过的相邻顶点，则任意选择一个走过去，并重复步骤 (2)；如果没有未走过的相邻顶点，就应该沿着搜索树向上回溯，并重复步骤 (2)。
  - (3) 当回溯到起点且无法继续走时，如果所有的顶点都已经走过，则遍历结束；否则转到步骤 (1)

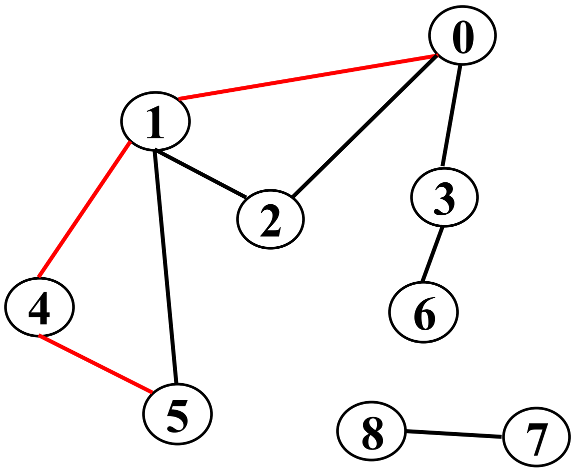
深度优先遍历：示例

- 对如下无向图进行深度优先遍历，一种可能的结果：
  – 首先选取顶点 0 开始遍历



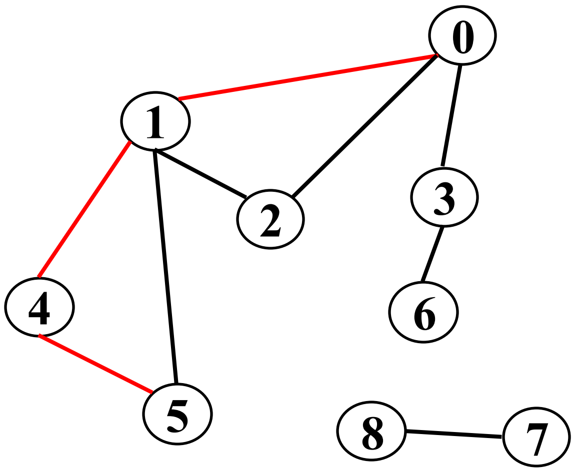
深度优先遍历：示例

- 对如下无向图进行深度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 不断访问未走到的顶点：0 – 1 – 4 – 5，此时结点 5 的邻居都已访问



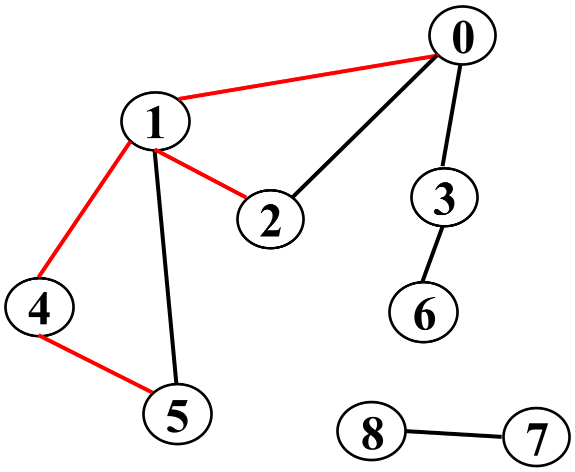
深度优先遍历：示例

- 对如下无向图进行深度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 不断访问未走到的顶点：0 – 1 – 4 – 5，此时结点 5 的邻居都已访问
  - 回溯至4，仍然是所有的邻居都已访问



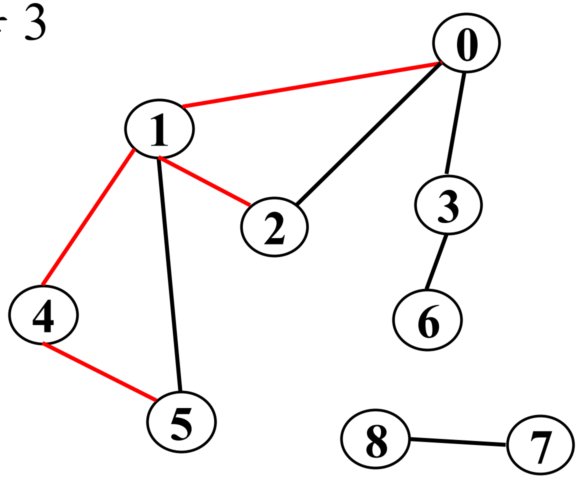
深度优先遍历：示例

- 对如下无向图进行深度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 不断访问未走到的顶点：0 – 1 – 4 – 5，此时结点 5 的邻居都已访问
  - 回溯至4，仍然是所有的邻居都已访问
  - 回溯至1，走到未访问的邻居 2，此时结点 2 的邻居都已访问



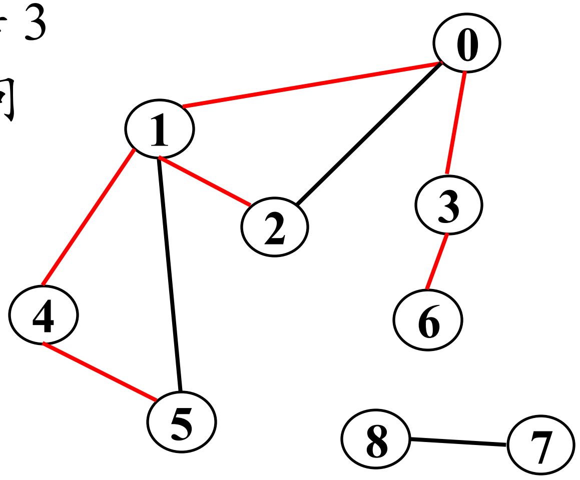
深度优先遍历：示例

- 对如下无向图进行深度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 不断访问未走到的顶点：0 – 1 – 4 – 5，此时结点 5 的邻居都已访问
  - 回溯至4，仍然是所有的邻居都已访问
  - 回溯至1，走到未访问的邻居 2，此时结点 2 的邻居都已访问
  - 再次回溯至1，然后回溯至0，存在未访问的邻居 3



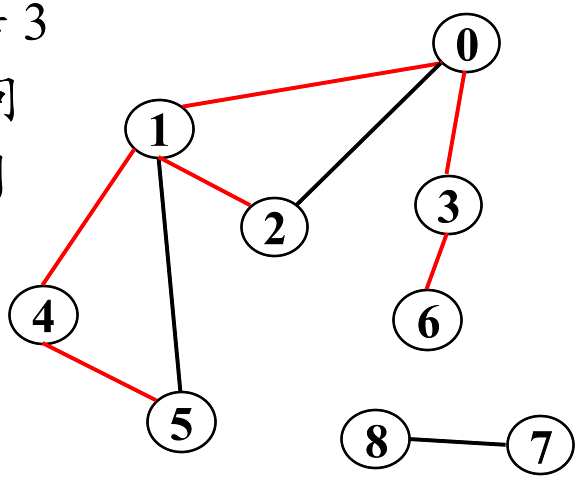
深度优先遍历：示例

- 对如下无向图进行深度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 不断访问未走到的顶点：0 – 1 – 4 – 5，此时结点 5 的邻居都已访问
  - 回溯至4，仍然是所有的邻居都已访问
  - 回溯至1，走到未访问的邻居 2，此时结点 2 的邻居都已访问
  - 再次回溯至1，然后回溯至0，存在未访问的邻居 3
  - 依次访问 3，6，此时结点 6 的所有邻居都已访问



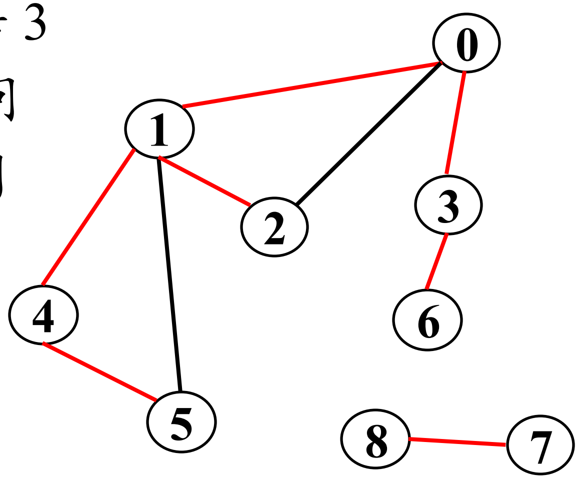
深度优先遍历：示例

- 对如下无向图进行深度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 不断访问未走到的顶点：0 – 1 – 4 – 5，此时结点 5 的邻居都已访问
  - 回溯至4，仍然是所有的邻居都已访问
  - 回溯至1，走到未访问的邻居 2，此时结点 2 的邻居都已访问
  - 再次回溯至1，然后回溯至0，存在未访问的邻居 3
  - 依次访问 3，6，此时结点 6 的所有邻居都已访问
  - 依次回溯至3，0，此时起点的所有邻居都已访问



深度优先遍历：示例

- 对如下无向图进行深度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 不断访问未走到的顶点：0 – 1 – 4 – 5，此时结点 5 的邻居都已访问
  - 回溯至4，仍然是所有的邻居都已访问
  - 回溯至1，走到未访问的邻居 2，此时结点 2 的邻居都已访问
  - 再次回溯至1，然后回溯至0，存在未访问的邻居 3
  - 依次访问 3，6，此时结点 6 的所有邻居都已访问
  - 依次回溯至3，0，此时起点的所有邻居都已访问
  - 选取顶点 8 继续遍历，访问邻居结点 7
  - 回溯至起点 8，所有结点都已访问，遍历结束



深度优先遍历：伪代码

```
# 对图 G 进行深度优先遍历的主函数
def dfs_traversal(G)
    将所有顶点设置成未访问
    对 G 中的所有顶点 v:
        如果 v 未访问，调用 dfs_visit(G, v)

# 给定起点进行深度优先遍历的子函数
def dfs_visit(G, v)
    访问结点v，并将其标记为已访问
    对于 v 的每个邻居节点 u:
        如果 u 未访问，调用 dfs_visit(G, u)
```

深度优先遍历

- 深度优先遍历的时间复杂度（假定采用邻接表实现）：
  - 初始化所有结点的时间复杂度为 O(n)
  - 主函数中，每个结点都会被检查一次
  - 子函数中遍历了结点的所有邻居。最坏情况下，每条边都可能被遍历一次
  - 时间复杂度为 O(n+e)

深度优先遍历：应用

- 思考：如何判断一个无向图是否连通，以及是否存在回路？

深度优先遍历：应用

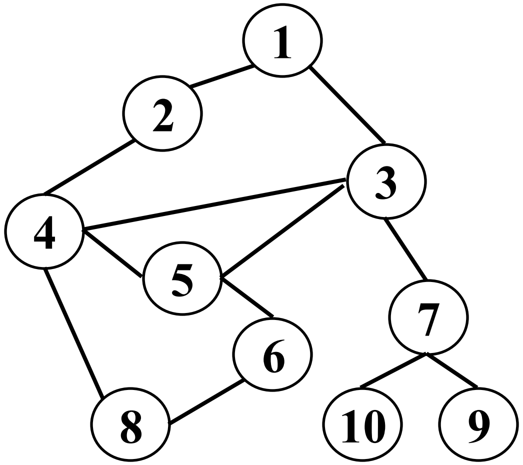
- 思考：如何判断一个无向图是否连通，以及是否存在回路？
- 连通性判断：
  – 任选一个顶点进行一次深度优先遍历的尝试（dfs_visit），如果能够访问到所有的顶点，就说明图是连通的。
- 回路判断：
  – 若图中不存在回路，则在深度优先遍历的过程中，对于当前顶点 v 的每个邻居 u，u 或者没有被访问过，或者是 v 在搜索树中的父节点。每前进一步都检查所有的邻居，若不满足上述性质，就说明图中存在回路。
  – 另一种思路：使用并查集。将每个顶点初始化为一个等价类；遍历所有的边，对每条边关联的两个顶点执行 Union 操作。如果某两个顶点在 Union 操作之前就已经属于同一等价类了，则说明存在回路。

广度优先搜索

- 给定图中的起点和终点，深度优先搜索（DFS）得到的路径不保证是最短的。
- 广度优先搜索能保证得到长度最短的路径。
  - 广度优先搜索，也称为宽度优先搜索
- 广度优先搜索的关键在于将顶点“分层”
  - 对应搜索树中结点的层次
  - 起点位于第 0 层，距离为 1 的顶点位于第一层，依此类推
  - 按层次从低到高扩展顶点，并用队列存放；先搜索层数较低的结点，后搜索层数较高的结点，从而保证能够找到最短路径
  - 为了记录最短路径，顶点入队时需要记录其在搜索树中的父结点

广度优先搜索：示例

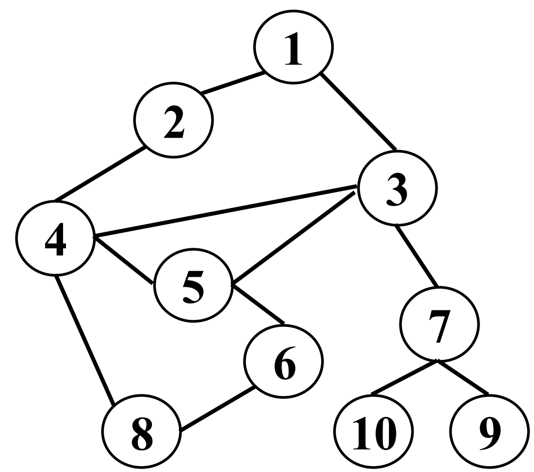
- 在下图中寻找由结点1到结点8的路径



广度优先搜索：示例

- 在下图中寻找由结点1到结点8的路径
  - 首先将结点 1 标记为已访问，并入队

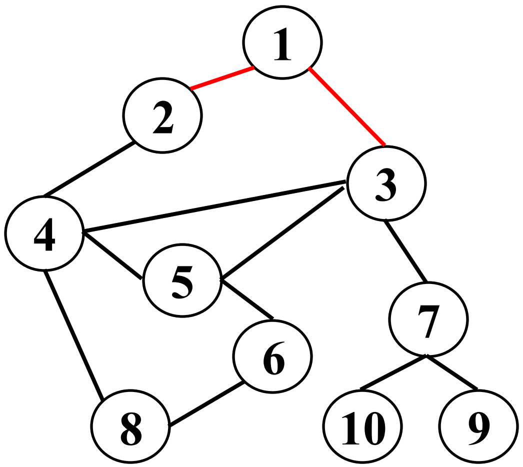
Q = [1]



广度优先搜索：示例

- 在下图中寻找由结点1到结点8的路径
  - 首先将结点 1 标记为已访问，并入队
  - 取出队头元素 1，入队并标记未访问的邻居

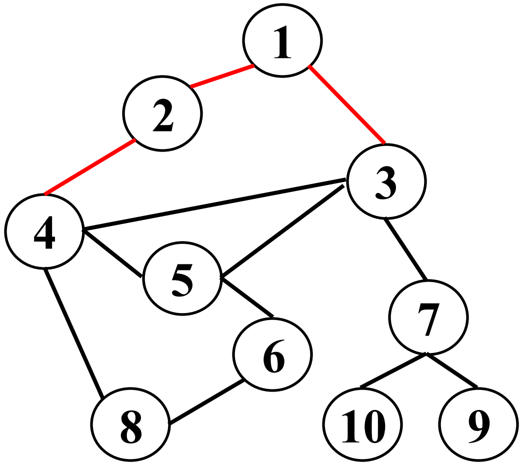
Q = [2, 3]



广度优先搜索：示例

- 在下图中寻找由结点1到结点8的路径
  - 首先将结点 1 标记为已访问，并入队
  - 取出队头元素 1，入队并标记未访问的邻居
  - 取出队头元素 2，入队并标记未访问的邻居

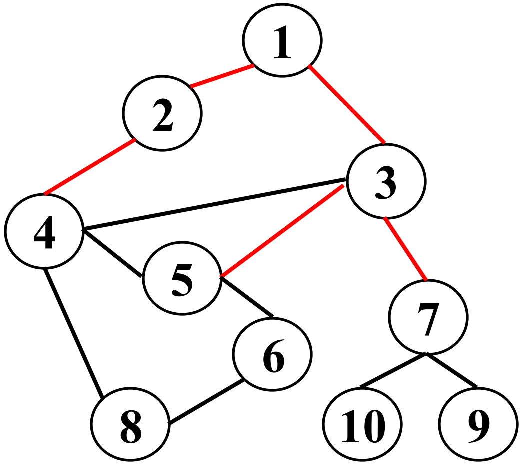
Q = [3, 4]



广度优先搜索：示例

Q = [4, 5, 7]

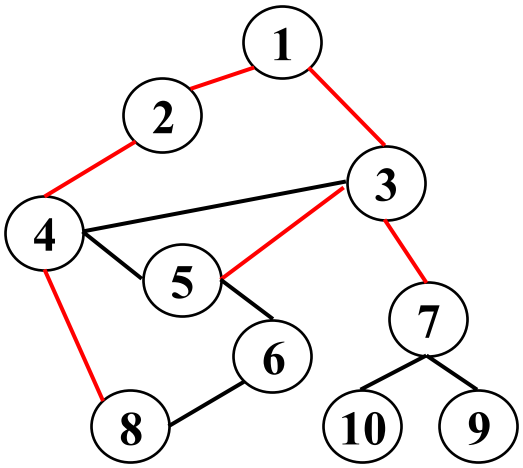
- 在下图中寻找由结点1到结点8的路径
  - 首先将结点 1 标记为已访问，并入队
  - 取出队头元素 1，入队并标记未访问的邻居
  - 取出队头元素 2，入队并标记未访问的邻居
  - 取出队头元素 3，入队并标记未访问的邻居



广度优先搜索：示例

- 在下图中寻找由结点1到结点8的路径
  - 首先将结点 1 标记为已访问，并入队
  - 取出队头元素 1，入队并标记未访问的邻居
  - 取出队头元素 2，入队并标记未访问的邻居
  - 取出队头元素 3，入队并标记未访问的邻居
  - 取出队头元素 4，入队并标记未访问的邻居

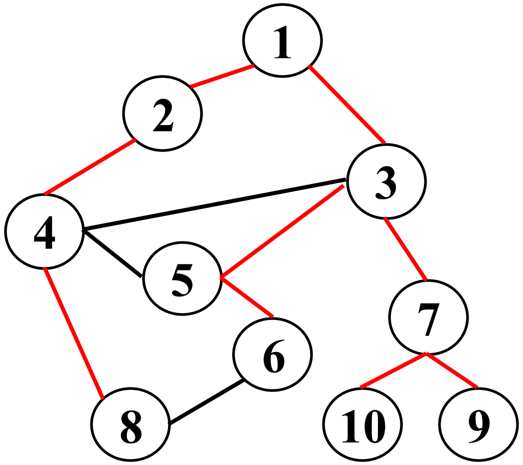
Q = [5, 7, 8]



广度优先搜索：示例

- 在下图中寻找由结点1到结点8的路径
  - 首先将结点 1 标记为已访问，并入队
  - 取出队头元素 1，入队并标记未访问的邻居
  - 取出队头元素 2，入队并标记未访问的邻居
  - 取出队头元素 3，入队并标记未访问的邻居
  - 取出队头元素 4，入队并标记未访问的邻居
  - 依次对队头元素 5，7，8 执行相同操作
  - 队头元素为 8 时，达到终点，搜索结束

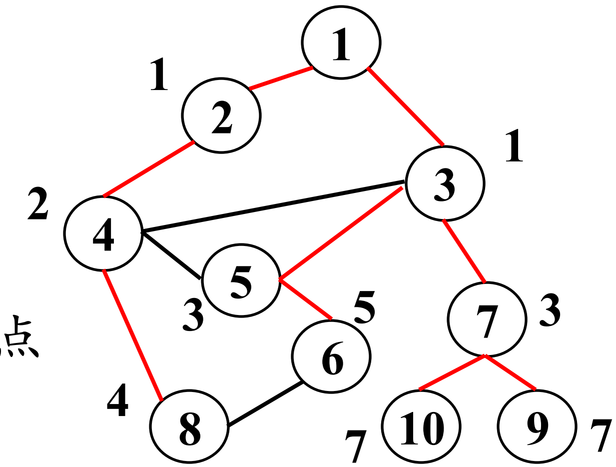
Q = [6, 10, 9]



广度优先搜索：示例

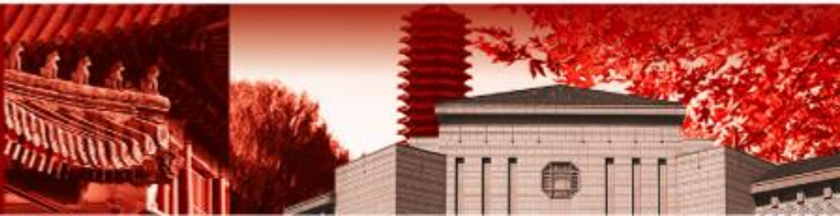
Q = [6, 10, 9]

- 在下图中寻找由结点1到结点8的路径
  - 首先将结点 1 标记为已访问，并入队
  - 取出队头元素 1，入队并标记未访问的邻居
  - 取出队头元素 2，入队并标记未访问的邻居
  - 取出队头元素 3，入队并标记未访问的邻居
  - 取出队头元素 4，入队并标记未访问的邻居
  - 依次对队头元素 5，7，8 执行相同操作
  - 队头元素为 8 时，达到终点，搜索结束
- 如何构建出由结点 1 到 8 的路径？
  - 顶点入队时，应该记录其再搜索树中的父节点
  - 从结点 8 逆向重构：8 – 4 – 2 – 1



广度优先搜索：伪代码

```
# 在图 G 中寻找一条由 start 到 end 的路径，不存在则返回 None
def bfs(G, start, end):
  将所有顶点初始化为未访问，父结点初始化为空，初始化队列为空
  将 start 标记为已访问，并将 start 入队
  while 队列非空:
    将队头元素赋值给 current_node
    if current_node == end：退出循环
    对 current_node 的每个邻居 v:
      若 v 未被访问，将 v 入队，并将 current_node 标记为已访问
      记录 v 的父节点为 current_node
  if end 结点未被访问：搜索失败 return None
  依据记录的每个结点的父结点，从终点开始逆向重构路径并返回
```



广度优先搜索：伪代码

```
# 在图 G 中寻找一条由 start 到 end 的路径，不存在则返回 None
def bfs(G, start, end):
    将所有顶点初始化为未访问，父结点初始化为空，初始化队列为空
    将 start 标记为已访问，并将 start 入队
    while 队列非空:
        将队头元素赋值给 current_node  ←
        if current_node == end：退出循环
        对 current_node 的每个邻居 v:
            若 v 未被访问，将 v 入队，并将 current_node 标记为已访问
            记录 v 的父节点为 current_node
    if end 结点未被访问：搜索失败 return None
    依据记录的每个结点的父结点，从终点开始逆向重构路径并返回
```

思考：是否可以改为在这里修改结点的访问状态

广度优先搜索

- 广度优先搜索的时间复杂度分析（假定采用邻接表实现）：
  - 初始化结点的代价为 O(n)
  - 迭代搜索过程中，每个结点至多入队 / 出队一次。因此，入队 / 出队的代价为O(n)
  - 对于队首元素，要对其所有邻居结点进行判断。该步骤中总的判断次数取决于图中边的总数，即代价为 O(e)
  - 重构路径的代价为 O(n)
- 广度优先搜索的时间复杂度为 O(n+e)

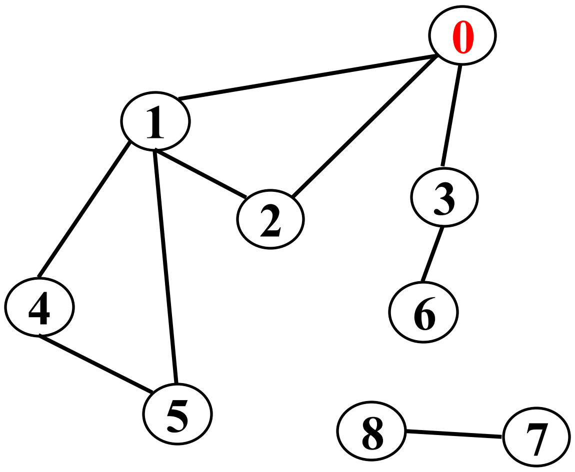
广度优先遍历

- 采用广度优先搜索的策略，遍历图中的所有顶点，称为广度优先遍历。具体步骤如下
  - (1) 在图中任意选择没有走过的顶点，作为遍历的起点，并访问之
  - (2) 依次访问图中与起点距离为 1 的顶点、距离为 2 的顶点等等，直到由起点可达的所有顶点都已经访问过。对于距离相同的顶点之间的先后次序则无要求。
    - 对图中的两个顶点 a, b，将 a 到 b 的距离定义为由 a 到 b 的所有路径中，边数最少的路径上的边数。
  - (3) 如果所有的顶点都已经走过，则遍历结束；否则转到步骤 (1)

广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历

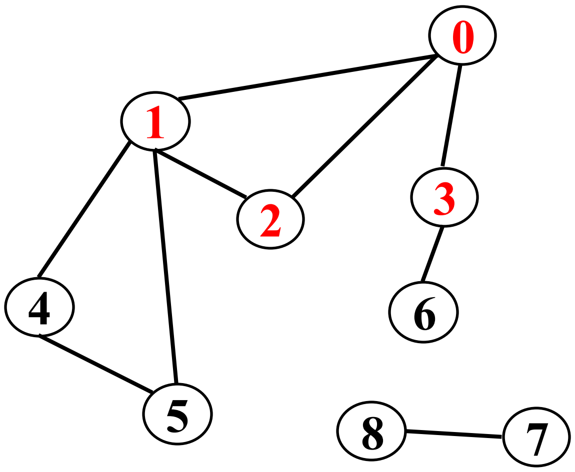
Q = [0]



广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队

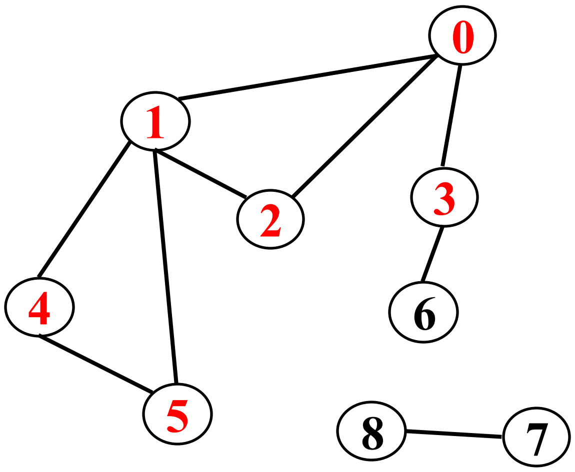
Q = [1, 2, 3]



广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队

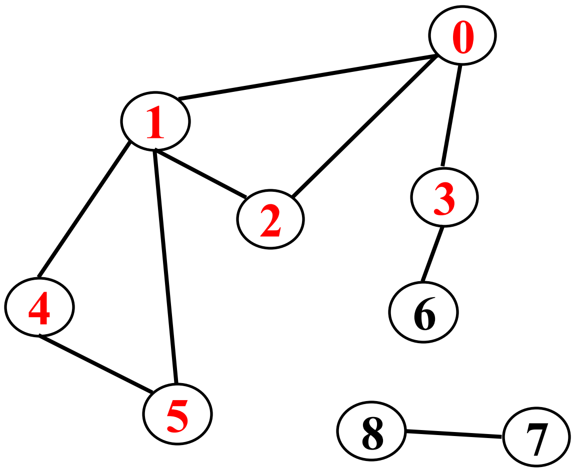
Q = [2, 3, 4, 5]



广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队
  - 访问顶点2，所有邻居都已访问

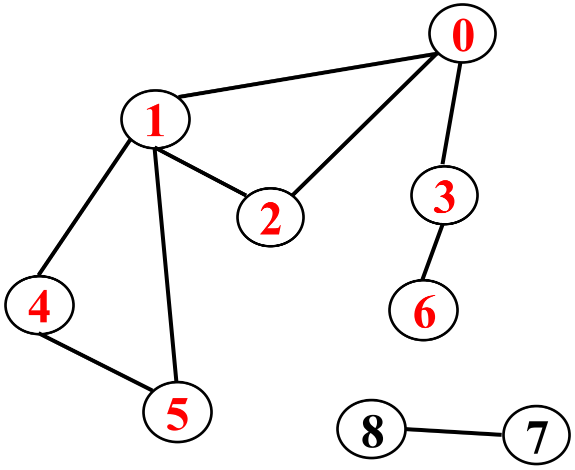
Q = [3, 4, 5]



广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队
  - 访问顶点2，所有邻居都已访问
  - 访问顶点3，并将顶点6 标记为已访问并入队

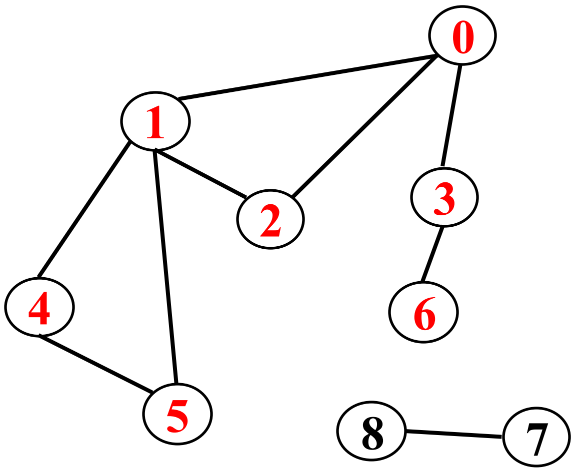
Q = [4, 5, 6]



广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队
  - 访问顶点2，所有邻居都已访问
  - 访问顶点3，并将顶点6 标记为已访问并入队
  - 访问顶点4，所有邻居都已访问

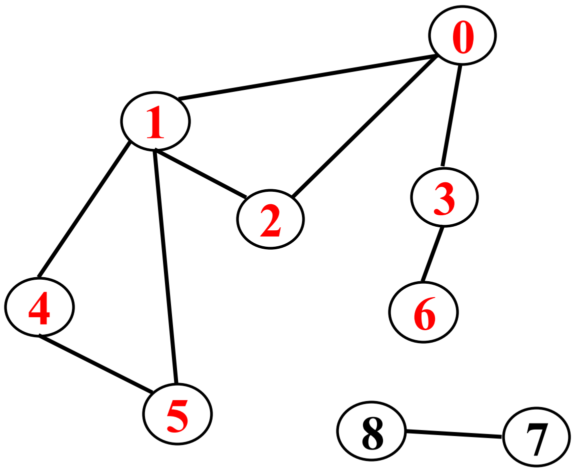
Q = [5, 6]



广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队
  - 访问顶点2，所有邻居都已访问
  - 访问顶点3，并将顶点6 标记为已访问并入队
  - 访问顶点4，所有邻居都已访问
  - 访问顶点5，所有邻居都已访问

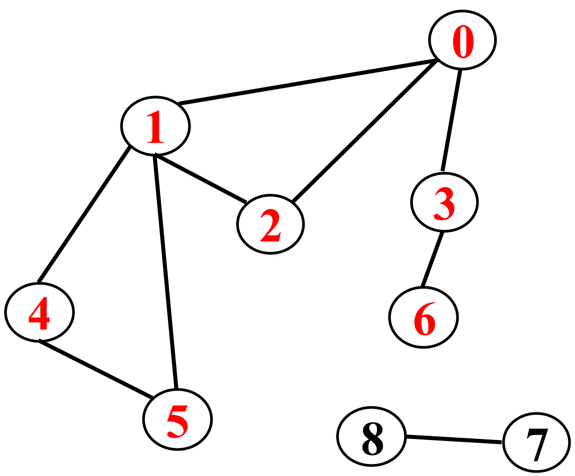
Q = [6]



广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队
  - 访问顶点2，所有邻居都已访问
  - 访问顶点3，并将顶点6 标记为已访问并入队
  - 访问顶点4，所有邻居都已访问
  - 访问顶点5，所有邻居都已访问
  - 访问顶点6，所有邻居都已访问

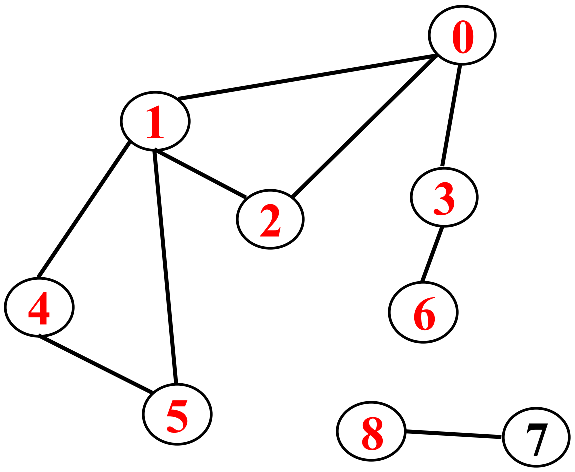
Q = []

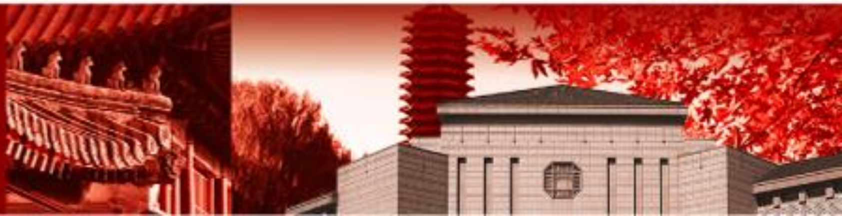


广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队
  - 访问顶点2，所有邻居都已访问
  - 访问顶点3，并将顶点6 标记为已访问并入队
  - 访问顶点4，所有邻居都已访问
  - 访问顶点5，所有邻居都已访问
  - 访问顶点6，所有邻居都已访问
  - 仍存在未访问结点，选取顶点 8 开始遍历

Q = [8]

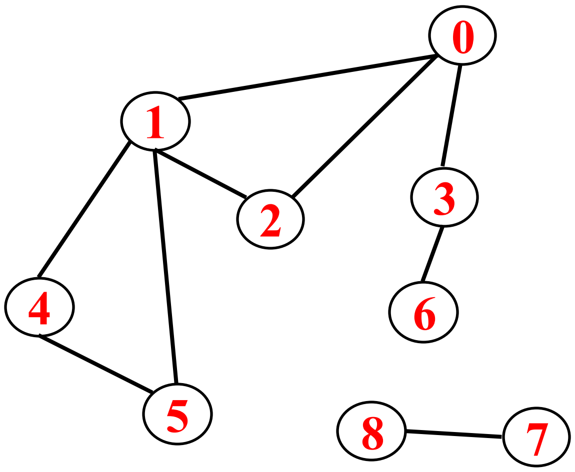




广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队
  - 访问顶点2，所有邻居都已访问
  - 访问顶点3，并将顶点6 标记为已访问并入队
  - 访问顶点4，所有邻居都已访问
  - 访问顶点5，所有邻居都已访问
  - 访问顶点6，所有邻居都已访问
  - 仍存在未访问结点，选取顶点 8 开始遍历
  - 访问顶点8，并将顶点 7 标记为已访问并入队

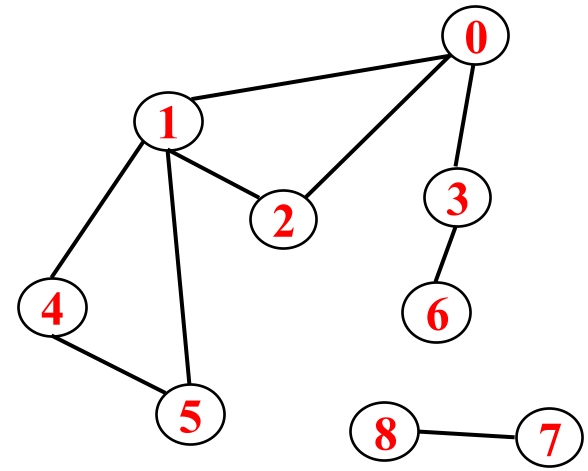
Q = [7]



广度优先遍历：示例

- 对如下无向图进行广度优先遍历，一种可能的结果：
  - 首先选取顶点 0 开始遍历
  - 访问顶点0，并将顶点1，2，3 标记为已访问并入队
  - 访问顶点1，并将顶点4，5 标记为已访问并入队
  - 访问顶点2，所有邻居都已访问
  - 访问顶点3，并将顶点6 标记为已访问并入队
  - 访问顶点4，所有邻居都已访问
  - 访问顶点5，所有邻居都已访问
  - 访问顶点6，所有邻居都已访问
  - 仍存在未访问结点，选取顶点 8 开始遍历
  - 访问顶点8，并将顶点 7 标记为已访问并入队
  - 访问顶点7，完成遍历

Q = [7]



广度优先遍历：伪代码

```
# 对图 G 进行广度优先遍历的主函数
def bfs_traversal(G)
  将所有顶点设置成未访问
  对 G 中的所有顶点 v：
    如果 v 未访问，调用 bfs_visit(G, v)

# 给定起点进行广度优先遍历的子函数
def bfs_visit(G, v)
  初始化队列，将 v 入队并标记位已访问
  while 队列非空：
    取出队头元素访问之
    对其所有未访问的邻居顶点，标记为已访问，并将其入队
```

广度优先遍历

- 广度优先遍历的时间复杂度：（假定采用邻接表实现）
  - 分析过程与广度优先搜索相同
  - 初始化结点的代价为 O(n)
  - 迭代搜索过程中，每个结点至多入队 / 出队一次。因此，入队 / 出队的代价为O(n)
  - 对于队首元素，要对其所有邻居结点进行判断。该步骤中总的判断次数取决于图中边的总数，即代价为 O(e)
- 总的时间复杂度同样为 O(n+e)

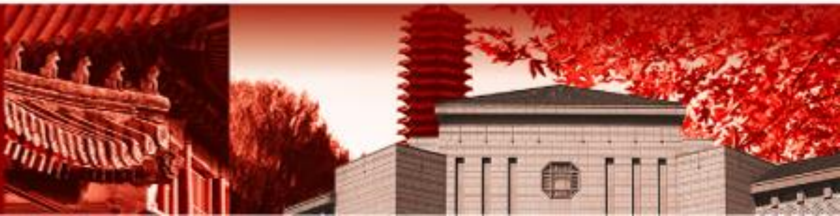
广度优先遍历：应用

- 案例：抓住那头牛
- 农夫知道一头牛的位置，想要抓住它。农夫和牛都位于数轴上，农夫的位置为 N，牛的位置为 K，且 0 ≤ N, K ≤ 100,000。农夫有以下两种移动方式
  - 从 X 移动到 X - 1 或 X + 1，用时 1 分钟
  - 从 X 移动到 2 * X，也用时 1 分钟

  农夫移动的过程中，牛保持位置不动。求农夫为了抓住牛所需要花费的最少时间

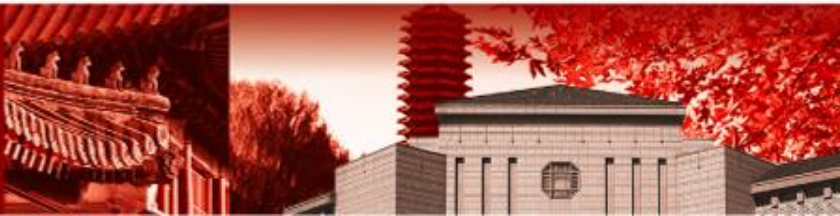
广度优先遍历：应用

- 问题建模：
  - 数轴上的一系列位置就是本题中的“状态”，即农夫位置的所有状态，对应到一个无穷大的无向图中的顶点集
  - 对于任意的 X，存在无向边 (X, X-1)，(X, X+1), (X, 2*X)
  - 本问题即为，求在该图中由顶点 N 至顶点 K 的最短路径长度
- 解决策略：使用广度优先搜索即可
  - 显然可以保证，对于任意的 N, K，从 N 到 K 一定是连通的
  - 只需要由农夫的位置 N 开始进行广度优先搜索，并记录所有顶点到顶点 N 的距离
  - 第一次访问到顶点 K 时，其距离即为答案



总结：图的搜索与遍历

- 图上的搜索操作与遍历操作是关系紧密的两种操作，有深度优先与广度优先两种基本策略。
  - 邻接表实现下，其时间复杂度都为 O(n+e)
- 许多问题可以建模为图上的搜索问题，使用 DFS/BFS解决。
- 深度优先搜索/遍历：
  - 特点是具有回溯过程；可以使用剪枝策略、启发式策略进行优化
  - 所需要的存储空间较少，主要是递归所需的栈空间
- 广度优先搜索/遍历：
  - 特点是优先遍历距离较近的顶点，能够搜索最短路径
  - 所需要的存储空间较大
- 无论采用哪一种策略，搜索 / 遍历时都需要进行判重操作。



课堂练习

- 倒水问题：假如你有一个 3 升的容器 A 和一个 5 升的容器 B，不借助其他测量工具的情况下，你该如何精确地量出 4 升水来呢？

课堂练习

- 倒水问题：假如你有一个 3 升的容器 A 和一个 5 升的容器 B，不借助其他测量工具的情况下，你该如何精确地量出 4 升水来呢？
  - 先把 A 装满水，全部倒入 B：(0, 3)
  - 再次把 A 装满水：(3, 3)
  - 用 A 中的水把 B 装满：(1, 5)
  - 倒空 B，然后将 A 中的水倒入 B：(0, 1)
  - 再次把 A 装满水，将 A 中的水全部倒入 B：(0, 4)

课堂练习

- 我们想要寻找一般倒水问题的解决方案：对于 n 个容积分别为 $V_1, V_2, \cdots\cdots, V_n$ 的容器，如何精确地量出 m 升水？（m 及所有容器的容积都为整数）
- 我们要求找出步骤最少的方法，可用的基本操作包括：
  - Fill：把容器装满水
  - Drop：把容器倒空
  - Pour：将一个容器 A 中的水倒入另一个容器 B，要求直到把 B 装满或者把 A 倒空
- $m \leq \max(V_1, V_2, \cdots\cdots, V_n)$
- 如果问题无解，算法应该识别并报告问题无解

解答

- 问题的状态即为，每个容器中包含的水的体积 L_i
  - 对应到图中的结点，用一个元组存储这一状态：$(L_1, L_2, \cdots, L_n)$
- 对应三种基本操作，结点之间存在三种有向边
- 有向边形如：$<(L_1, L_2, \cdots, L_n), (L'_1, L'_2, \cdots, L'_n)>$
  - Fill(i)：L_i 变为 V_i
  - Drop(i)：L_i 变为 0
  - Pour(i, j)：倾倒体积为 amount = min(L_j, V_i-L_i)，L_i, L_j 分别减少和增加 amount
- 原问题转变为搜索问题：起点为 (0, 0, ……, 0)，终点为有容器恰好包含 m 升水的任意结点。

解答

- 由于要找出最少步骤的方案，考虑使用广度优先搜索算法
- 判重操作
  – 对于数据规模较小的情况，可以记录所有可能的结点的访问状态，总结点数量为（V_1+1）*（V_2 +1）* ……（V_n+1）；如果数据规模较大，我们可以设置 visited 集合，用于存放已访问的结点。
- 重构路径
  – 对于数据规模较小的情况，可以记录所有结点在搜索树中的父节点，搜索结束后重构路径；如果数据规模较大，我们在每个结点上附加当前的操作序列。终点的操作序列即为所求方案。
- 判断解的存在性
  – 如果队列为空时，仍未到达任何目标结点，就表示已经穷尽了所有可能也没有解决问题，可以报告问题无解。