6. 图-1

- 图：基本概念和性质，基本操作
- 图的遍历：宽度优先，深度优先
- 图的表示和 Python 实现
- 生成树问题，DFS 生成树，BFS 生成树
- 最小生成树问题，Prim 算法，Kruskal 算法
- 最短路径问题，单源点最短路径和 Dijkstra 算法，所有顶点之间的最短路径和 Floyd 算法
- AOV 网，拓扑排序
- AOE 网，关键路径

图（Graph）

- 图是一种数学结构，数学里有分支“图论”，研究这种拓扑结构
- 数据结构课程把图看着一类复杂数据结构，用于表示具有各种复杂关系的数据集合。图在实际中应用很广泛（如第一章的例子）
- 图的结构比较复杂，因此有许多可能的实现方式，许多典型算法。本章介绍图的基本知识，基本实现方法，若干基本计算问题和重要算法
- 重点算法（这些算法是本章最重要的内容）：
  - 图的深度优先搜索与广度优先搜索
  - 最小生成树的 Prim 算法和 Kruskal 算法
  - 求单源最短路径的 Dijkstra 算法
  - 求所有顶点对之间最短路径的 Floyd 算法
  - 拓扑排序
  - 关键路径

图：基本概念

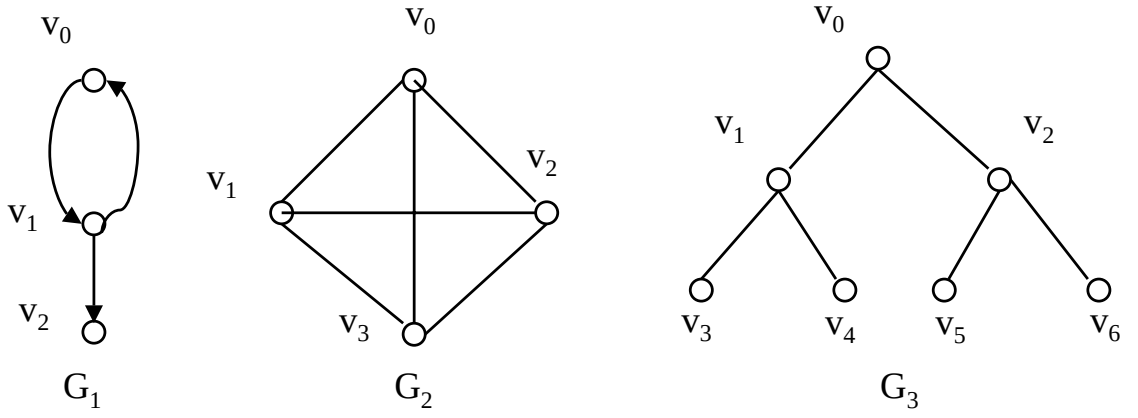
- 一个图是一个二元组 G = (V, E)，其中
  - V 是顶点的非空有穷集合（也可以有空图的概念，但用处不大）
  - E 是顶点偶对 (称为边) 的集合，$E \subseteq V \times V$
  - V 中的顶点也称为图 G 的顶点，E 中的边也称为图 G 的边
- 有向图：有向图中的边有方向，边是顶点的有序对

  有序对用尖括号表示。$<v_i,v_j>$ 表示从 v_i 到 v_j 的有向边，v_i 称为边的始点，v_j 是边的终点。$<v_i, v_j>$ 和 $<v_j, v_i>$ 表示两条不同有向边
- 无向图：无向图中的边没有方向，是顶点的无序对

  无序对用圆括号表示，(v_i , v_j) 和 (v_j , v_i) 表示同一条无向边
- 如果图 G 里有边 $<v_i,v_j> \in E$（或 $(v_i, v_j) \in E$）
  - 顶点 v_j 称为 v_i 的邻接顶点（对无向图，邻接点是双向的）
  - 这种边称为与顶点 v_i 相关联的边或 v_i 的邻接边

图的图示

- 与有序树类似，图中的邻接边也可以规定或不规定顺序
- 图可用图形自然表示（圆圈表示顶点，线段或箭头表示边）



- 两个限制：1）不考虑顶点到自身的边，若 (v_i ,v_j) 或 $<v_i ,v_j>$ 是 G 的边，则要求 $v_i \neq v_j$；2）顶点间没有重复出现的边，若 (v_i ,v_j) 或 $<v_i ,v_j>$ 是 G 的边，则它是这两个顶点间的唯一边（不存在多重边）
- 去掉这些限制得到稍微不同的数学对象，也可以研究它们

图：概念和性质

- 完全图：任意两顶点之间都有边的图（有向图或无向图）。显然：
  - n 个顶点的无向完全图有 n*(n-1)/2 条边
  - n 个顶点的有向完全图有 n*(n-1) 条边
- 注意一个重要事实：$|E| \leqslant |V|^2$，也即 $|E| = O(|V|^2)$

  也就是说，边的条数可能达到顶点数的平方
- 度（顶点的度）：与一个顶点邻接的边的条数

  对有向图，度数还进一步分为入度和出度，分别表示以该顶点为始点或者终点的边的条数
- 无论是有向图还是无向图，顶点数 n，边数 e 和度数满足下面关系：

  其中 $D(v_i)$ 表示顶点 v_i 的度数

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

路径和相关性质

- 路径：对于图 G = (V, E)，如果存在顶点序列 $v_{i0},v_{i1},\ldots,v_{i(m)}$，使得 (v_{i0},v_{i1})，$(v_{i1},v_{i2}),\ldots$，$(v_{i(m-1)}, v_{i(m)})$ 都在 E 中（都是 G 的边，对于有向图是 $<v_{i0},v_{i1}>$，$<v_{i1},v_{i2}>$，…，$<v_{i(m-1)}, v_{i(m)}>$ 都在 E 中）
  - 则说从顶点v_{i0} 到$v_{i(m)}$ 存在一条路径
  - $<v_{i0},v_{i1},\ldots,v_{i(m)}>$ 称为是从 v_{i0} 到 $v_{i(m)}$ 的（一条）路径
- 路径长度：路径上边的条数
- 回路（环）：起点和终点相同的路径

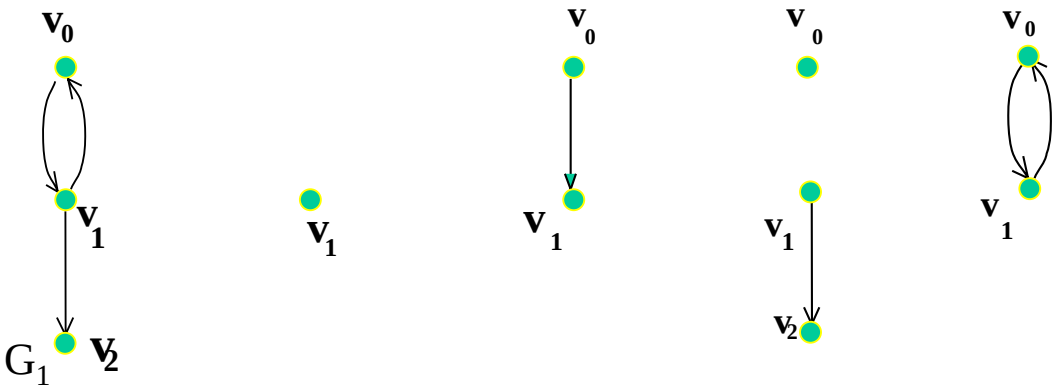
  如果除起点和终点外的其他顶点均不相同，则称为简单回路
- 简单路径：内部不包含回路的路径

  即，路径上的顶点除起点和终点可能相同外，其它顶点均不相同

  易见，简单回路也是简单路径

子图，有根图

- 对图 G = (V，E) 和 G' = (V'，E')，如果 V' ⊆ V 且 E' ⊆ E，就称 G' 是 G 的子图。特别的，G 是其自身的子图
- 下面是有向图 G_1 的几个子图（包括其自身）



- 有根图：如果在有向图 G 里存在一个顶点 v，从 v 到图 G 中其它每个顶点均有路径，则称 G 为一个有根图，称 v 为图 G 的一个根

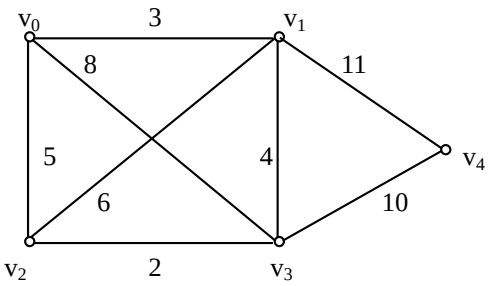
  有根图中的根可能不唯一

连通和连通图

- 连通：如果无向图 G 中有从 v_i 到 v_j 的路径（显然也是从 v_j 到 v_i 的路径)，则称 v_i 与 v_j 连通（顶点 v 到自身默认连通，有向图的定义类似）
- 无向图 G 的连通性
  - 连通图：如果 G 中任意两个不同顶点 v_i 和 v_j 之间都连通，则称 G 为一个连通图
  - G 的极大连通子图（连通分量）G' 是 G 的连通子图，且 G 中没有真包含 G' 的连通子图。若 G 不连通，则其连通分量多于一个
  - G 的连通分量形成了 G 的一个划分
- 有向图 G 的连通性
  - 强连通图：如果 G 中任意两个不同顶点 v_i 和 v_j 之间都有路径（注意路径有方向），则称 G 是一个强连通图
  - 强连通分量：G 的极大强连通子图称为它的强连通分量
  - G 的强连通分量形成其顶点的一个划分（注意与无向图不同）

带权图和网络

- 若图 G 的每条边都被赋以一个权值，G 称为是带权图
  - 权值可用于表示实际应用中与顶点关联有关的某些信息
  - 带权的连通无向图称为网络
- 下图为一个网络



- 网络是实际应用非常广泛的一类图结构

  下面会看到网络的一些重要性质和算法，也可以看到它们的应用

图的基本操作

- 作为复杂的数据结构，图上可能定义许多操作。下面列举一些操作
  - 创建图，创建空图，或基于顶点和边的数据创建图
  - 顶点的个数（**vertex_num**）和边的条数
  - 所有顶点的集合 **vertices**，所有边的集合 **edges**
  - 增加一条边 **<v1,v2>** 或 **(v1,v2)**，**add_edge (v1 , v2)**；是否存在边 **<v1,v2>** 或 **(v1,v2)**，**get_edge(g , v1 , v2)**
  - 顶点 **v** 的入度和出度（结果用二元序列表示），**vdegree (v)**
  - 找出顶点 **v** 相邻边（集合或表），如出边 **out_edges(v)**
- 遍历操作。与树遍历的最重要差异是：

  需要防止再次进入已经遍历过的部分

  需要考虑图的连通性问题（如果图不连通，遍历完一个连通分支并不是整个图遍历的结束，还需要继续遍历其他连通分支）

图的表示

- 图的结构比较复杂，任意两个顶点间都可能存在边
  - 需要表示顶点及顶点间的边，存在很多很多可能的方法
  - 应该根据具体应用和需要做的操作等选择图的表示方法
- 图的最基本表示方法是邻接矩阵表示法（下面讨论）
- 表示图中邻接关系的矩阵通常非常稀疏，空间浪费可能很大。很多图中边数与顶点数成线性关系，如中国铁路线路图。为降低空间代价，人们提出了许多其他方法，它们都可看着邻接矩阵的“压缩”版，如：
  - 邻接表表示法
  - 邻接多重表表示法
  - 图的十字链表表示，等

  在 Python 语言里实现图，也有许多可能的方法
- 下面先介绍邻接表表示法

图的表示

- 邻接矩阵就是一个表示图中顶点间邻接关系的方阵

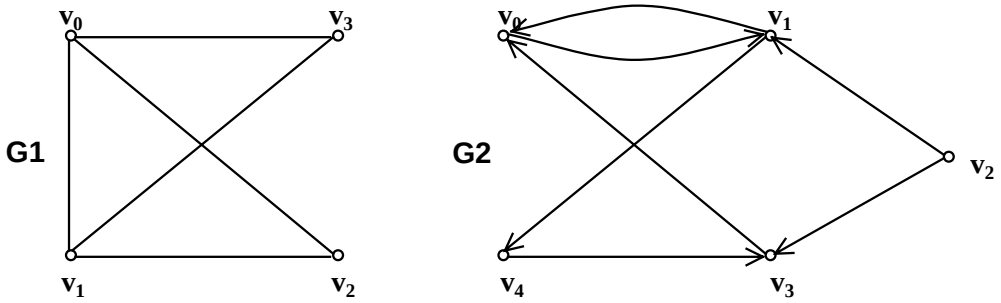
  设 G = (V，E) 为 n 个顶点的图，其邻接矩阵为如下的矩阵 A：

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

- 邻接矩阵只表示图中顶点个数和顶点间联系（边）
  - 每个顶点对应一个矩阵下标
  - 通过一对下标可以确定图中一条边的有无
- 如果顶点本身还有其他信息，就需要用另外的方式表示顶点信息
  - 例如，可以考虑另外用一个顶点表
  - 可以用与矩阵同样下标引用同一顶点的表元素

邻接矩阵的例

- 下面无向图 G1 和有向图 G2 的邻接矩阵分别为邻接矩阵 A1 和 A2



$$A_1=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix}\qquad A_2=\begin{bmatrix}0&1&0&0&0\\1&0&0&0&1\\0&1&0&1&0\\1&0&0&0&0\\0&0&0&1&0\end{bmatrix}$$

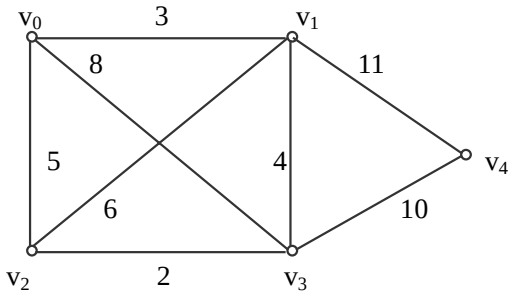
无向图的邻接矩阵总是对称矩阵

带权图的邻接矩阵表示

- 带权图的边附带有用信息，用邻接矩阵表示时，可以把权信息记录在矩阵里。矩阵元素是边的权值，无边的情况另行表示
- 设 G 是带权图，w_{ij} 是边 (v_i,v_j) 或 $<v_i,v_j>$ 的权，图 G 的邻接矩阵定义为（用邻接矩阵元素记录边的权）：

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

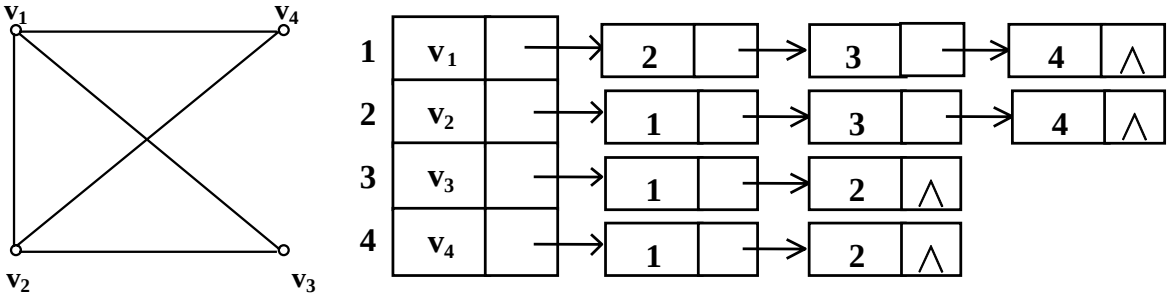
下面带权图的两个邻接矩阵表示 A_3 和 A_4，根据需要用 0 或∞表示无边



$$A_3=\begin{bmatrix}0&3&5&8&0\\3&0&6&4&11\\5&6&0&2&0\\8&4&2&0&10\\0&11&0&10&0\end{bmatrix}\qquad A_4=\begin{bmatrix}\infty&3&5&8&\infty\\3&\infty&6&4&11\\5&6&\infty&2&\infty\\8&4&2&\infty&10\\\infty&11&\infty&10&\infty\end{bmatrix}$$

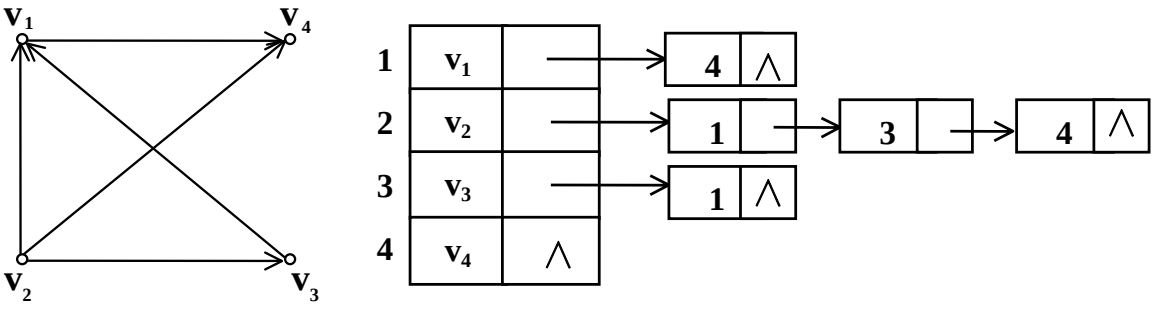
图的邻接表表示法

- 邻接表表示的形式与树的“子表表示”类似，用
  - 一个顶点表表示图中顶点及其相关信息
  - 每个顶点关联着一个邻接表，记录其邻接顶点
- 无向图：
  - 一个顶点表，每个顶点有一个邻接边的表
  - 边 (v_i,v_j) 在顶点 v_i，v_j 的边表各有一结点，在边表中存储两次
  - 顶点 v_i 的边表中结点个数为顶点 v_i 的度数



图的邻接表表示法

- 有向图
  - 一个顶点表，每个顶点关联着一个边表
  - 顶点 vi 的边表里每个结点对应的是以 vi 为始点的一条边，因此，可以把这种边表称为出边表
  - 顶点 vi 的边表中结点个数为顶点 vi 的出度
- 也可采用记录入边的表（入边表），表中结点个数是顶点的入度



出边表

在 Python 里实现图

- 在 Python 里实现图数据结构可以有许多方法，一些例子：
- 用 list 的 list（或 tuple 的 tuple）直接实现邻接矩阵

  使用方便，判邻接简单，但存储代价较大，不适合较大的图
- 用字典从下标序对映射到邻接边的值。检索效率高，适合稀疏矩阵（邻接矩阵常如此）。字典实现复杂，基于连续表示，处理大矩阵需试验
- 用 Python 内置的 Bytes 字节向量类似或标准库的 array 类型
  - 前者存储效率高，可能用于实现很大的图，但编程复杂一些
  - 后者可以表示整数或浮点权值的带权图，编程也比较复杂
- 自定义类型实现邻接表表示。下面考虑两种简单技术
- 下面假定有一个全局变量 infinity 表示无穷大，如果带权图中 v 到 v' 无边，就以 infinity 作为边的权值。可定义

  infinity = float("inf") # infinity 的值是 float 类型

图的 Python 实现

- 现在先考虑基于邻接矩阵实现图的问题
  - 矩阵元素可以是 1 或权值表示有边，用一个特殊值表示“无联”
  - 假设用户提供一个 unconn 值，默认为 0
  - 构造函数基于给定的矩阵参数建立一个图
- 定义一个类表示这种图，参数包括一个邻接矩阵，做一个拷贝：

```
class Graph: # basic graph class, using adjacent matrix
    def __init__(self, mat, unconn = 0):
        vnum1 = len(mat)
        for x in mat:
            if len(x) != vnum1: # Check square matrix
                raise ValueError("Argument for 'GraphA' is bad.")
        self.mat = [mat[i][:] for i in range(vnum1)]
        self.unconn = unconn
        self.vnum = vnum1
```

邻接矩阵实现

- 继续：

```
def vertex_num(self): return self.vnum

def add_edge(self, vi, vj, val = 1):
    assert 0 <= vi < self.vnum and 0 <= vj < self.vnum
    self.mat[vi][vj] = val

def add_vertex(self):
    raise AdjGraphError(
        "Adjacent Matrix does not support 'add_vertex'")

def get_edge(self, vi, vj):
    assert 0 <= vi < self.vnum and 0 <= vj < self.vnum
    return self.mat[vi][vj]
```

- 这个邻接矩阵实现不支持增加结点

  如果要支持增加结点，还需要每行增加元素，比较麻烦但也可以做

邻接矩阵实现

```
def out_edges(self, vi):
    assert 0 <= vi < self.vnum
    return self._out_edges(self.mat, vi, self.unconn)

@staticmethod
def _out_edges(mat, vi, unconn):
    edges = [ ]; row = mat[vi]
    for i in range(len(row)):
        if row[i] != unconn: edges.append((i, row[i]))
    return edges

def __str__(self):
    return "[\n" + "\n".join(map(str, self.mat)) + "\n]\n"\
          + "Unconnected: " + str(self.unconn)
```

- 把生成邻接结点表的内部方法定义为静态方法，可能在其他地方使用
- 定义 __str__ 函数，内置函数 str(...) 调用它生成 Graph 对象的字符串表示。对象的字符串表示可用于存入文件或输出（print）

压缩的邻接矩阵（邻接表）实现

- 邻接矩阵表示的缺点是空间复杂性与结点数的平方成正比
  - 以中国的铁路路线图为例，大约 6000 个车站，如果用邻接矩阵表示，矩阵元素约 3600 万个
  - 大部分车站入度/出度为 2/2，总边数不超过 3 万（出入边分别计）
  - 99.9% 以上的矩阵元素是 infinity（或者 0）
- 考虑一种“压缩的”表示形式
  - 每个顶点的邻接边用一个（不一定等长的）list 表示，元素形式为 (vj, w)，vj 是边的终点，w 是边的信息（权）
  - 整个图就是这种 list 的 list，其中每个元素对应一个顶点
  - 对这种表示，很容易给已有的图添加顶点（增加一项）
  - 可以采用与前面 Graph 同样的接口，定义同样名字的方法
  - 内部表示采用新的一套，一些方法可以继承

邻接表（压缩的邻接矩阵）实现

新类的定义：

```
class GraphA(Graph):
    def __init__(self, mat, unconn = 0):
        vnum1 = len(mat)
        for x in mat:
            if len(x) != vnum1: # Check square matrix
                raise ValueError("Argument for 'GraphA' is bad.")
        self.mat = [Graph._out_edges(mat, i, unconn)
                    for i in range(vnum1)]
        self.vnum = vnum1
        self.unconn = unconn

    def add_vertex(self): # For new vertex, return a new index
        self.mat.append([])
        self.vnum += 1
        return self.vnum-1 # 返回新顶点的编号
```

邻接表（压缩的邻接矩阵）实现

```
def add_edge(self, vi, vj, val = 1):
    assert 0 <= vi < self.vnum and 0 <= vj < self.vnum
    row = self.mat[vi]
    for i in range(len(row)): # insert in order
        if row[i][0] == vj:       # replace a value for mat[vi][vj]
            self.mat[vi][i] = (vj, val)
            return
        if row[i][0] > vj: break
    else:
        i += 1 # adjust for the new entry at the end
    self.mat[vi].insert(i, (vj, val))

def get_edge(self, vi, vj):
    assert 0 <= vi < self.vnum and 0 <= vj < self.vnum
    for i, val in self.mat[vi]:
        if i == vj: return val
    return self.unconn
```


邻接表（压缩的邻接矩阵）实现

```
def out_edges(self, vi):
    assert 0 <= vi < self.vnum
    return self.mat[vi]
```

- 注意：这里用顺序检索插入/访问，结点的出度很小时操作的效率不是问题。如果结点出度可能很大，可以考虑用二分检索，或为每个结点关联一个 **dict**（从邻接结点到边值的字典），以支持快速插入和访问
- 上面两个类的实现接口相同但其内部实现不同：
  - 从两个类的对象出发能调用的方法完全一样
  - 数据表示不同，操作的实现方法不同，操作的时间和空间效率也不同（下面会看到这些对算法效率的影响）
  - 如果基于这套方法定义操作图的函数，实现图上的重要算法，对这两种不同的图表示都适用
- 这里只表示了边信息，顶点就是一个编号。如果顶点有更多信息，例如名字，相关数据等，可以在图对象里扩充一个顶点表或顶点字典

图算法

- 很多实际问题可以归结为图和图上的计算，例如：
  - 网络路由（我们每天都在用）
  - 集成电路（和印刷电路板）的设计和布线
  - 运输和物流中的各种规划安排问题
  - 工程项目的计划安排
  - 许多社会问题计算，如金融监管（例如关联交易检查）
- 一旦从应用中抽象出“图”，应用问题的解决就可能变成图算法问题
- 下面将讨论一些图上的算法
  - 这些算法都有很清晰的应用背景
  - 很多算法里蕴涵着巧妙的想法和理论根据，其正确性需要（也可以）严格证明，复杂性非常重要（需要处理的问题规模可能很大）
  - 实现中常常用到前面讨论的一些数据结构

图的遍历

- 遍历：按某种方式系统访问图中所有顶点且每个顶点仅访问一次的过程

  也称为图的周游。前面讨论的状态空间中的状态和相邻关系就形成了一个图，图的遍历对应于一次穷尽的状态空间搜索
- 遍历的基本部分是访问一顶点所在连通分支（对有向图是该结点的可达子图）的全部顶点。如果图不连通，还要考虑对其他部分的访问

  基本的图遍历方法同样是深度优先遍历和宽度优先遍历
- 深度优先遍历，也就是按照深度优先搜索（**Depth-First Search**）的方式遍历，做法是：
  - 从指定顶点 **v** 出发，先访问 **v**（并将其标记为已访问）
  - 从 **v** 的未访问过的邻接顶点出发进行深度优先搜索（邻接顶点可能排一种顺序），直到与 **v** 连通的所有顶点都已访问（递归）
  - 如果图中还存在未访问顶点，则选出一个未访问顶点，由它出发重复前述过程，直到图中所有顶点都已访问为止

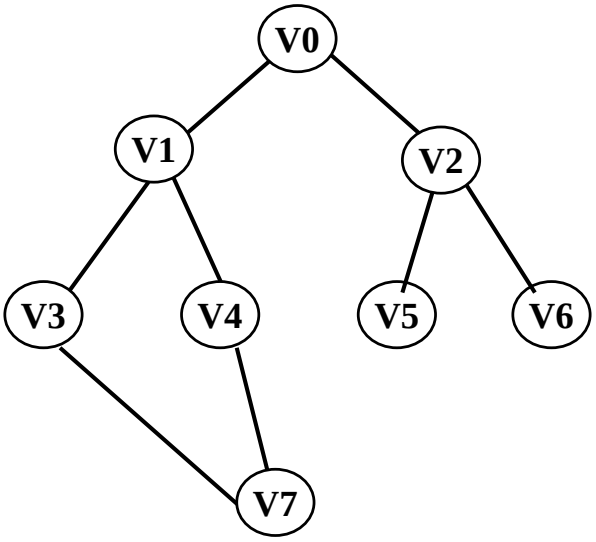
图的遍历

- 通过深度优先遍历顺序得到的顶点序列称为该图的深度优先搜索序列（Depth-First-Search 序列），简称为 DFS 序列
  - 对邻接点的不同访问顺序将得到不同的 DFS 序列（不唯一）
  - 如果规定了图中各顶点的邻接点顺序，就确定了 DFS 序列
- 广度优先遍历通过广度优先搜索（Breadth-First Search）方式遍历，具体过程是：
  - 从指定顶点 v 出发，先访问 v 并将其标记为已访问
  - 依次访问 v 的所有相邻顶点 $v_1, v_2, \ldots, v_m$ （可规定顺序），再依次访问与 $v_1, v_2, \ldots, v_m$ 邻接的所有未访问顶点，直到所有顶点都访问
  - 如果图中还存在未访问顶点，则选择一个未访问顶点，由它出发进行广度优先搜索，直到所有顶点都已访问为止
- 通过广度优先遍历得到的顶点序列称为图中顶点的广度优先搜索序列（Breadth-First Search 序列）或BFS序列

  如果规定了图中个顶点的邻接点顺序，BFS 序列就确定了

图的遍历

- 看一个简单例子



- 对右图从 V1 开始遍历，邻接点访问按从左到右的顺序
- 深度优先遍历得到的 DFS 序列：<V0, V1, V3, V7, V4, V2, V5, V6>
- 宽度优先遍历得到的 BFS 序列：<V0, V1, V2, V3, V4, V5, V6, V7>

深度优先遍历：非递归算法

- 非递归深度优先遍历算法，与迷宫求解类似。遍历从 0 可达的顶点集：

```
def DFS_graph(graph, v0):
    vnum = graph.vertex_num()
    visited = [0]*vnum   # visited 记录访问过的顶点
    visited[v0] = 1; DFS_seq = [v0]      # DFS 序列作为遍历结果
    st = SStack()
    st.push((0, graph.out_edges(v0)))  # 入栈 (i, edges), 说明下次
    while not st.is_empty():                # 访问的是边是 edges[i]
        i, edges = st.pop()
        if i < len(edges):
            v, e = edges[i]
            st.push((i+1, edges)) # 下次回到这里将访问 edges[i+1]
            if not visited[v]:       # v 是未访问顶点，访问并记录
                DFS_seq.append(v)
                visited[v] = 1
                st.push((0, graph.out_edges(v))) # 下面访问的边组
    return DFS_seq
```


深度优先遍历：非递归算法

- 现在考虑算法的复杂性
- 时间复杂性
  - 构造 visited 数组和建立 DFS_seq 的时间都是 O(|V|)
  - 算法过程中进栈出栈操作的次数对应于图的边数，O(|E|) 时间
  - 对于 Graph 对象，取所有出边的操作总耗时是 $O(|V|^2)$，对于第二种图实现，取出边总耗时是 O(|V|)
  - 综合起来，对于 Graph 对象（图的邻接矩阵实现），遍历的时间复杂性是 $O(|V|^2)$；对于 GraphA 对象（图的链接表实现）；时间复杂性是 O(max(|V|, |E|)
- 空间复杂性
  - visited 和 DFS_seq 都需要 O(|V|) 空间，栈的深度不超过结点个数，所以算法的空间复杂性是 O(|V|)
- 从这里可以看到数据结构的不同实现方法对算法复杂性的影响

生成树：概念

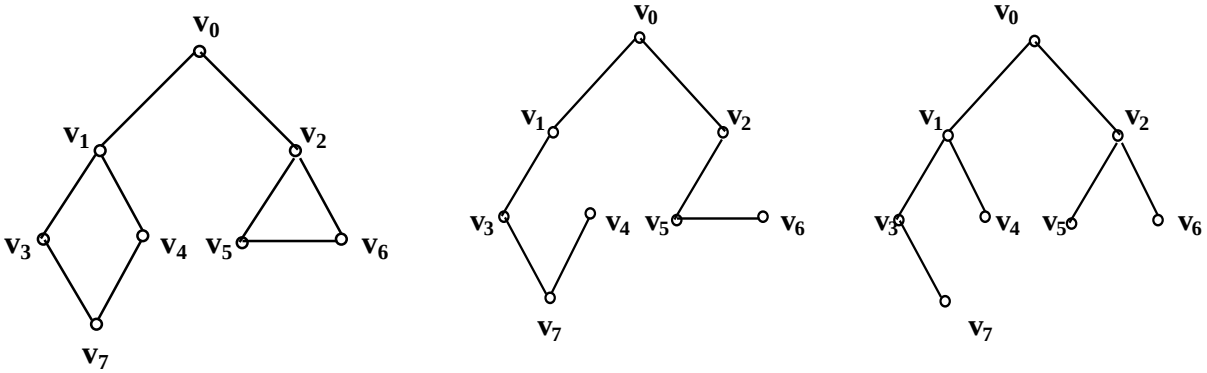
- 生成树：从连通无向图或强连通有向图 G 的任一顶点 v_0 出发，或从有根有向图的根 v_0 出发，存在到图中其他各结点的路径
  - 若 G 有 n 个顶点，必然可找到 n-1 条边，其中包含了从 v_0 到其他所有结点的路径（容易通过数学归纳法证明）。这样 n-1 条边（加上 G 所有顶点）形成了 G 的一个子图 T
  - 无向图 G 的子图 T 是一个最小连通子图（去掉任意一条边后不再连通）。n 个顶点 n-1 条边的图成树形，因此称 T 为 G 的生成树。对有向图的定义类似。图的生成树可能不唯一
- 性质：n 个顶点的连通图的生成树包含恰好 n-1 条边

  无向树就是连通的无环图。有向树中所有的边都位于从根到其他结点的（有方向的）路径上
- 非连通的无向图划分为一组连通分量，可以定义它们的生成树林

  n 个顶点 m 个连通分量的图 G 的生成森林恰好包含 n-m 条边

遍历和生成树

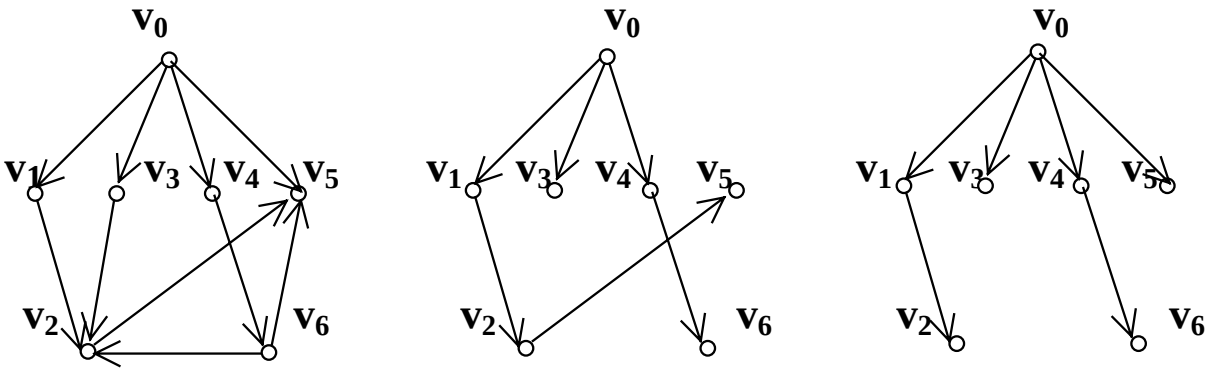
- 从连通无向图或强连通有向图中任一顶点出发遍历，或从有根有向图的根顶点出发遍历，都可以访问到所有顶点
  - 遍历经过的边加上所有顶点，就构成该图的一棵生成树
  - 构造生成树的过程可以按深度优先遍历或广度优先遍历
- 遍历中记录访问的所有顶点和经过的边，就得到深度优先生成树，或广度优先生成树，简称为 DFS 生成树或 BFS 生成树



无向图　　DFS生成树　　BFS生成树

遍历和生成树

- 有向图的遍历产生的生成树



- 也可以把连通无向图的生成树定义为包含其全部顶点的最小连通子图，一般无向图的生成树林定义为各连通分量的最小连通子图的集合
- 也可以把强连通有向图的生成树定义为包含了其全部顶点的最小有根子图（一棵有向树，去掉一条边就不再是有根图了）
- 生成树描述了图的一种框架结构，也称为支撑树

构造 DFS 生成树

- 构造 **DFS** 生成树，需要设法表示生成树上的边
  - 注意：生成树上的边形成了从初始顶点 **v0** 到其他顶点的路径
  - 在这一簇路径里，一个顶点可能有多个“下一”顶点

    但每个顶点至多有一个“前一”顶点
  - 一种可能方式是记录这种“前一顶点”关系

    遍历的所有顶点之后，根据前一顶点关系就能追溯出有关路径了
- 下面算法里用了一个包含 **vnum** 个元素的表 **span_forest**
  - **span_forest[vi]** 的形式是序对 **(vj, e)**
  - 其中 **vj** 是从 **v0** 到 **vi** 的路径上 **vi** 的前一顶点，**e** 是该边的信息

    虽然 **0/1** 信息可以不记，但被处理的图中的边可能标着权值
- 在主函数里做循环，是为了找到其他尚未访问的顶点（从前面已经考虑过的起始遍历顶点出发不可达的顶点）

构造 DFS 生成树：递归算法

- 递归构造 DFS 生成森林。由于后面需要路径的路由，在压栈的项里必须记录出边的始点。另外，visited 表也不需要了

```
def DFS_span_tree(graph):
    vnum = graph.vertex_num()
    span_forest = [None]*vnum
    def dfs(graph, v):          # 递归遍历函数，在递归中记录经由边
        nonlocal span_forest    # 要操作 nonlocal 变量 span_forest
        for u, w in graph.out_edges(v):
            if span_forest[u] is None:
                span_forest[u] = (v, w)
                dfs(graph, u)

    for v in range(vnum):
        if span_forest[v] is None:
            span_forest[v] = (v, 0)
            dfs(graph, v)
    return span_forest
```


DFS 算法的复杂性分析

- 时间复杂性，情况与遍历算法类似：
  - 在递归访问过程中，表 span_forest 的每个顶点设置一次，不会重复设置。设置的次数与递归调用的次数一样（两个语句顺序执行）。但 dfs 里的循环可能访问到图中每条边一次
  - 对于 Graph 对象，取所有出边需要 $O(|V|^2)$ 时间
  - 综合起来可知，对图的邻接矩阵实现，算法的时间复杂性是$O(|V|^2)$；对图的链接表实现，算法的时间复杂性是 $O(max(|V|, |E|))$
- 空间复杂性：
  - span_forest 需要 $O(|V|)$ 空间，函数递归的深度不会超过 $O(|V|)$。所以这个算法需要 $O(|V|)$ 存储空间
- 基于 BFS 搜索的遍历算法，另外基于两个递归或非递归 DFS 搜索的遍历算法可以自己实现，作为课下的算法设计和编程练习