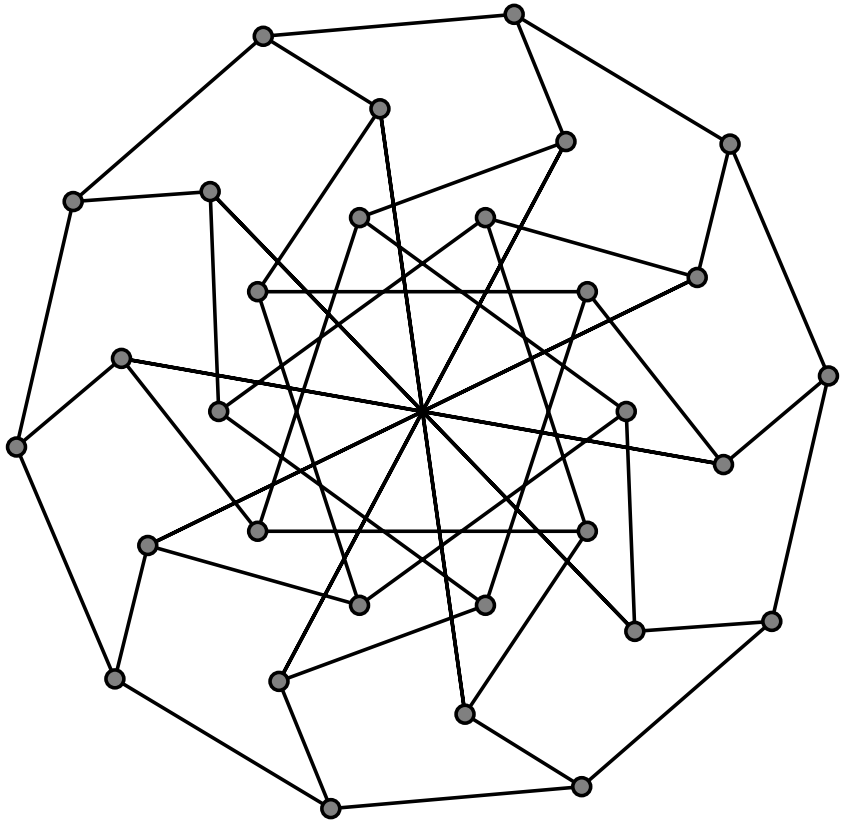
数据结构

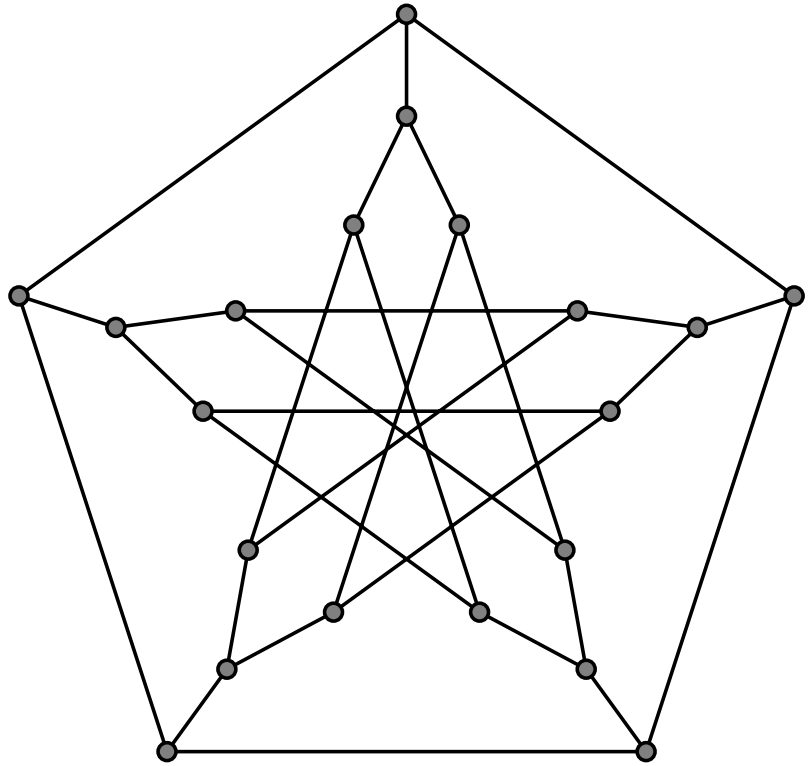
夏天

xiat@ruc.edu.cn

中国人民大学信息资源管理学院

Graph





Content

- 图的定义
- 图的存储表示
- 图的遍历
- 图的连通性

图的用途

- 用数学语言来说，网络通常被视为一种加权图（Weighted Graph），其中节点（Node）代表实体，边（Edge）代表实体间的关系，而边的权重（Weight）则代表了这种关系的某种度量（如距离、成本、流量等）。
- 除了数学上的定义，网络还具有实际的物理含义，它是从相同类型的实际问题中抽象出来的模型。这些实际问题包括但不限于交通系统、物流运输、通信系统等，因此习惯上会根据其应用领域称之为相应的网络类型，如交通网络、运输网络、通信网络等。
- 现实生活中的许多问题，如交通路线的优化、ERP（企业资源规划）系统的工作计划制定等，都可以借助图与网络的模型进行建模和求解。这些模型能够帮助我们更好地理解和分析问题的本质，从而找到有效的解决方案。

社会网络 (social network)

- 社会网络是指由社会个体成员之间因互动而形成的相对稳定的关系体系。
- 社会网络分析（SNA）作为社会学的一个重要分支，自上世纪 30 年代末起逐步发展，成为研究社会结构的重要方法和技术。从 30 年代到 60 年代，这一领域吸引了来自心理学、社会学、人类学以及数学、物理、统计、概率等领域的学者，他们共同推动了 SNA 的理论、方法和技术的发展，使其成为研究社会结构的重要范式。

六度分隔 (Six Degrees of Separation) I

- 六度空间理论（也称为小世界现象或六度分隔理论）最初是在 1929 年由匈牙利作家弗里奇斯·卡林思（Frigyes Karinthy）在他的短篇小说中提出的，他提出世界上所有人都可以通过至多五个中间人联系起来。
- 1967 年，哈佛大学的社会心理学教授 Stanley Milgram (1933-1984) 进行了一项著名的实验来验证这一理论。他设计了一个连锁信件实验，从 Nebraska 和 Kansas 随机选择了 300 多名志愿者，请他们寄一封信函给指定的一名住在波士顿的股票经纪人。实验结果表明，有六十多封信函最终到达了目标人手中，并且这些信函在传递过程中经过的中间人的数目平均只有大约 5 个。这一发现验证了六度空间理论，并使其受到了全世界的广泛关注。

六度分隔 (Six Degrees of Separation) II

- 此外，Milgram 还发现了漏斗效应，即大部分的信件传递都是由少数几个“明星人物”完成的，这些人在社交网络中具有很高的连接性，能够更快地传递信息。
- 六度空间理论不仅激发了学术界的深入研究，还催生了电影《六度空间》（又称《六度分离》或《连锁反应》）等文化产品，进一步增强了公众对这一理论的兴趣和认知。

150 法则 (Rule of 150) I

- 从欧洲发源的“赫特兄弟会”是一个自给自足的农民自发组织，他们有一个不成文的严格规定：每当聚居人数超过 150 人的规模，他们就把它变成两个，再各自发展。
- 150 也被称为“邓巴数字 (Dunbar’s number)”，该理论认为人的大脑新皮层大小有限，提供的认知能力只能使一个人维持与大约 150 人的稳定人际关系。
- 把人群控制在 150 人以下似乎是管理人群的一个最佳和最有效的方式。早期中国移动的动感地带 sim 卡最多保存 150 个手机号，早期一个 MSN 帐号只能对应 150 个联系人。

150 法则 (Rule of 150) II

- 不管你认识多少人，或者与更多人建立了弱链接，那些强链接仍然在此次此刻符合 150 法则。这也符合“二八”法则，即 80% 的社会活动可能被 150 个强链接所占有。

网络模型 I

- 随机网络：
  - 在交通网络中，随机网络可以模拟道路拥堵的情况。假设道路的连接（箭线）和交叉口（节点）的实现具有一定的随机性，这取决于交通流量、道路状况等多种因素。
  - 用途：随机网络用于描述那些节点和连接具有不确定性的系统。在交通网络中，随机网络模型可以帮助预测和规划交通流量，优化交通设计，减少拥堵。此外，随机网络还广泛应用于计算机科学、社会学、生物学等领域，用于模拟和分析各种复杂系统的行为。

网络模型 II

- 小世界网络
  - 社交网络可以看作是一个小世界网络。在社交网络中，虽然大多数人只与少数人保持紧密联系（短路径），但几乎每个人都可以通过少数几个熟人联系到网络中的任何其他人（小世界现象）。
  - 用途：小世界网络模型用于描述那些具有较短的平均路径长度和较高的聚类系数的网络。这种网络结构使得信息在网络中传播得非常快，因此小世界网络模型在信息传播、社交网络分析、推荐系统等领域具有广泛应用。此外，小世界网络还用于研究生物网络、神经网络等复杂系统的行为。

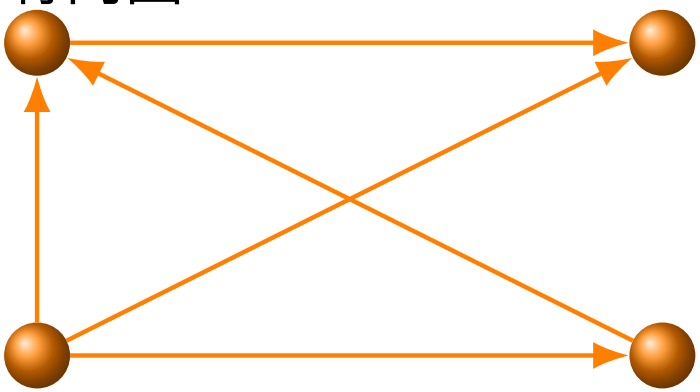
网络模型 III

- 无标度网络
  - 互联网可以看作是一个无标度网络。在互联网中，少数大型网站（Hub 点）拥有大量的链接，而大多数网站只有少量的链接。这种网络结构使得互联网呈现出无标度性，即节点的连接数（度数）分布严重不均匀。
  - 用途：无标度网络模型用于描述那些具有严重不均匀分布特性的网络。在互联网中，无标度网络模型有助于理解网络的拓扑结构、稳定性和鲁棒性等特性。此外，无标度网络还广泛应用于社交网络、金融网络、生物网络等领域的研究，以揭示这些复杂系统的内在规律和运行机制。

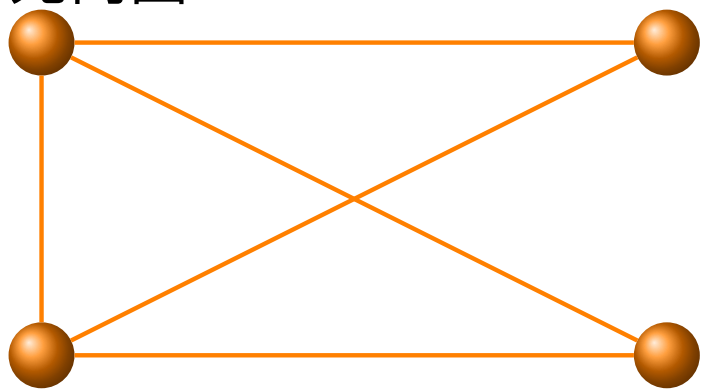
图 (Graph)

- 图 $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, V 是顶点 (Vertex) 集合,E 是边/弧 (Edge/Arc) 的集合.
- 顶点的度、出度和入度

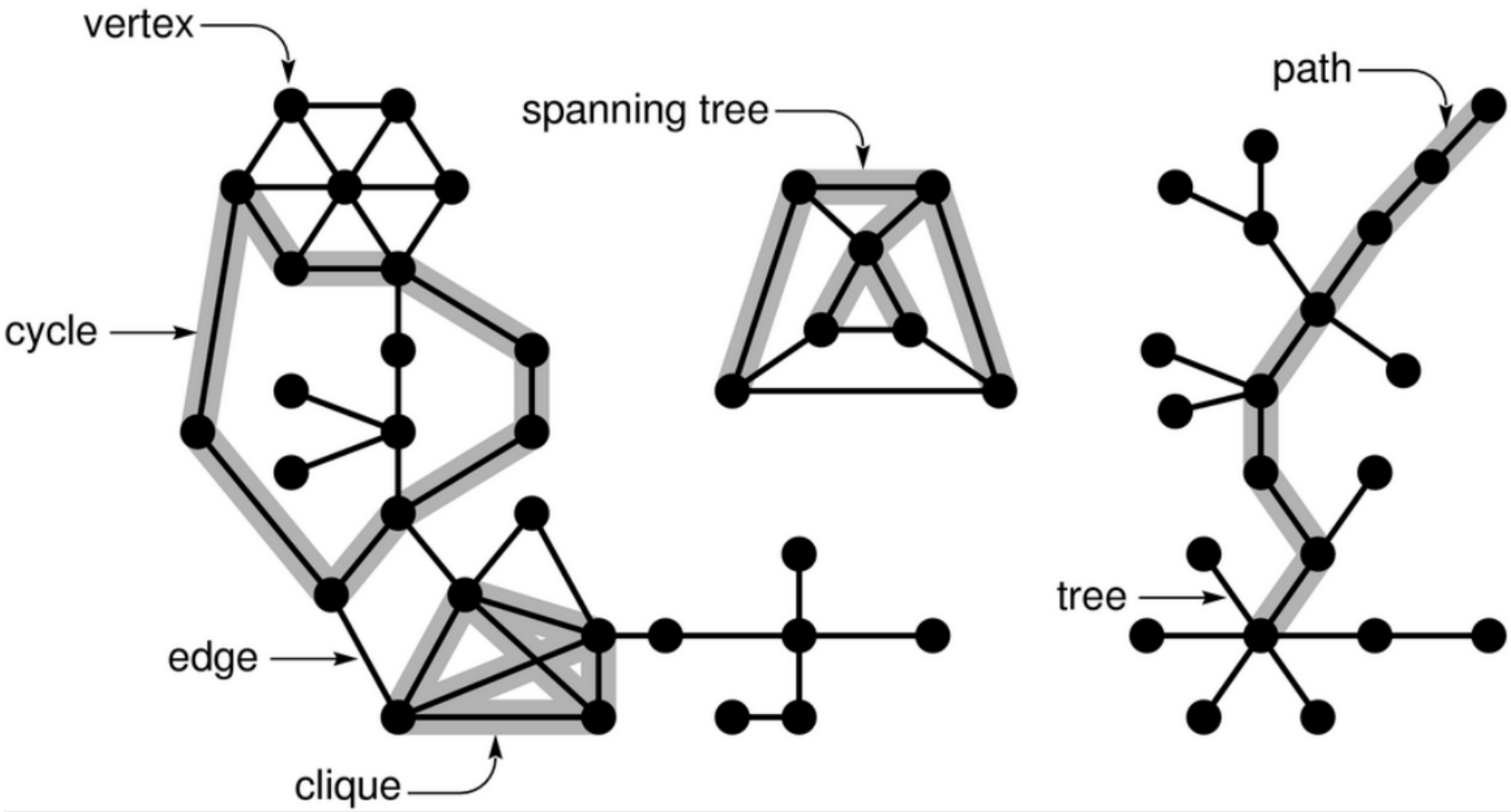
有向图:



无向图:



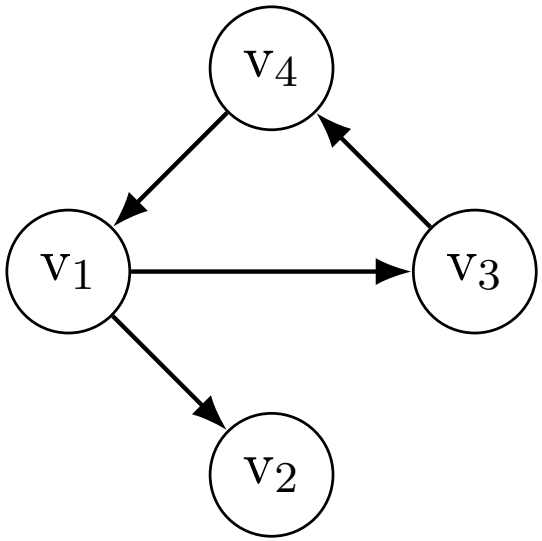
图的相关概念



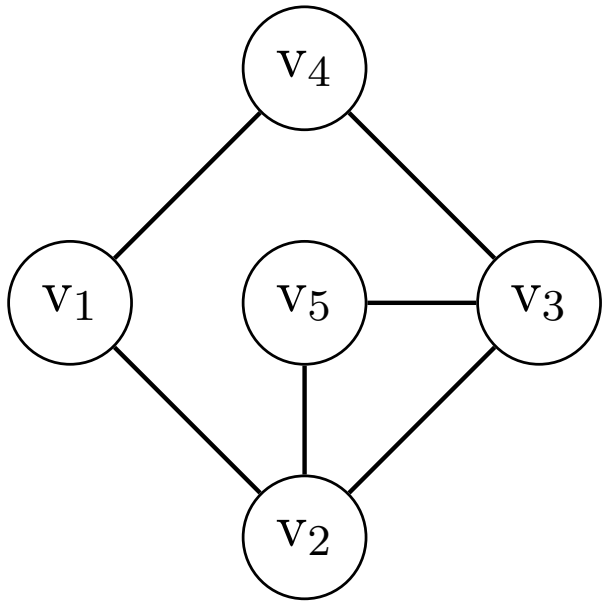
图的存储

如何表达下图的信息?

有向图:



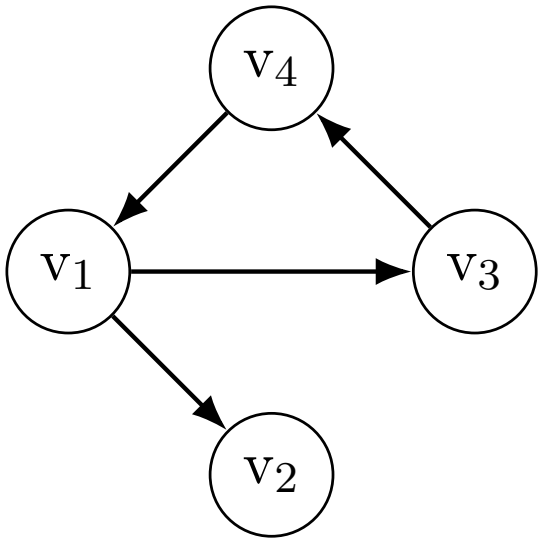
无向图:



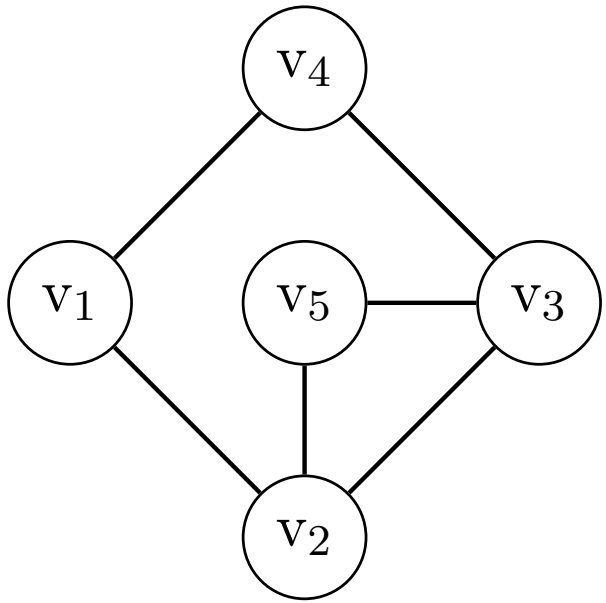
图的存储

如何表达下图的信息?

有向图:

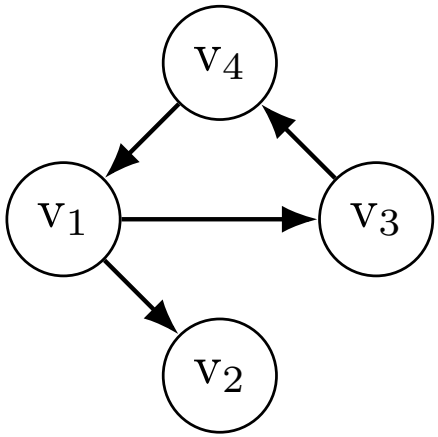


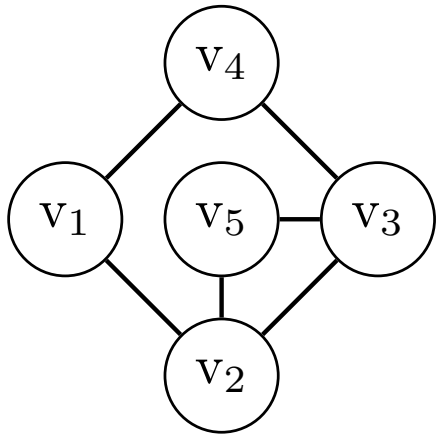
无向图:



- 可用邻接矩阵表达顶点及其关系。

图的存储



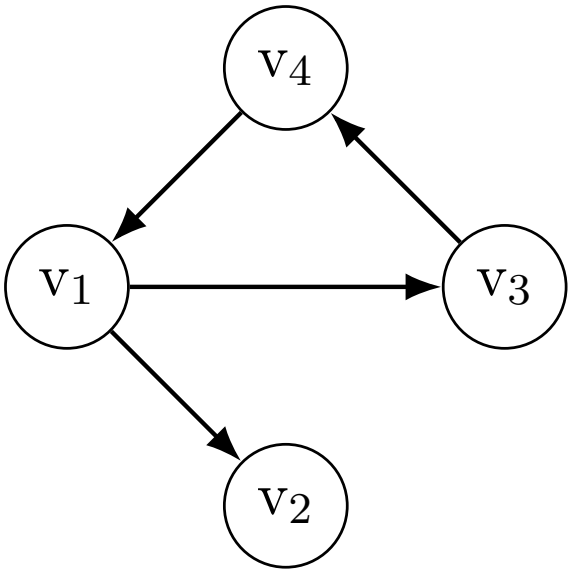


$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

- 根据邻接矩阵，如何判断各顶点的度？

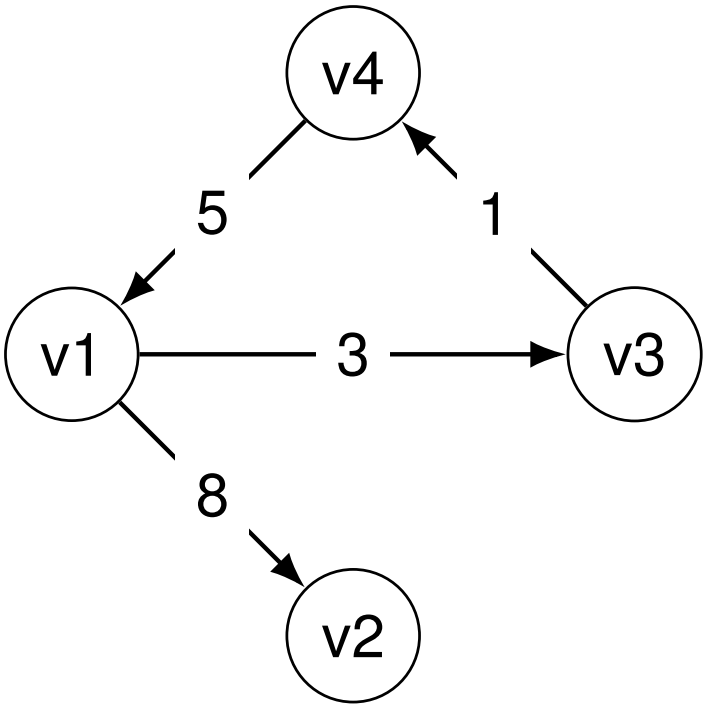
有向图的连续存储方式：邻接矩阵



- 建立二维数组 $\mathrm{A[n][n]}$, $\mathrm{n} = |\mathrm{V}|$
- 另需存放 n 个顶点信息

$$
\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}
\begin{array}{c}
\begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\
\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}
\end{array}
$$

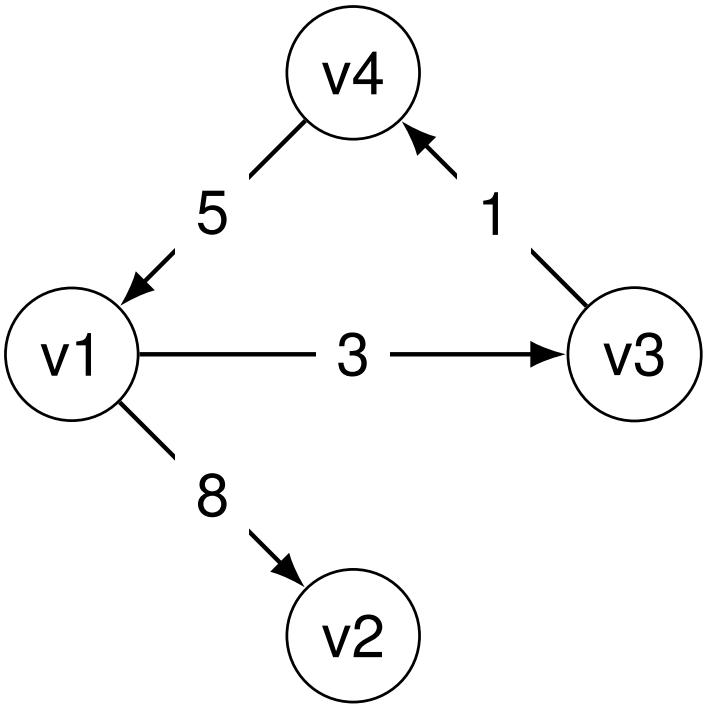
网的邻接矩阵



$$
\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}
\begin{array}{c}
\begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\
\begin{pmatrix} \infty & 8 & 3 & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 1 \\ 5 & \infty & \infty & \infty \end{pmatrix}
\end{array}
$$

- 有些图的边带有权重 (常用来表示成本、距离、时间等), 这样的图称为：网。
- 网的邻接矩阵表达权重, 没有边的顶点之间的权重默认为 ∞
- 邻接矩阵表示方法非常直观、简单, 但是会有什么问题?

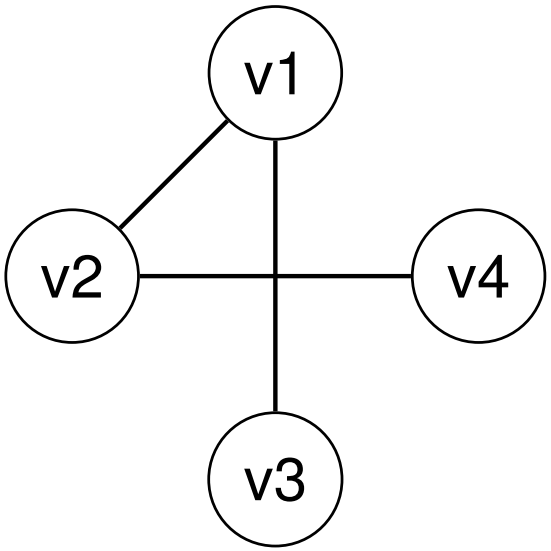
网的邻接矩阵

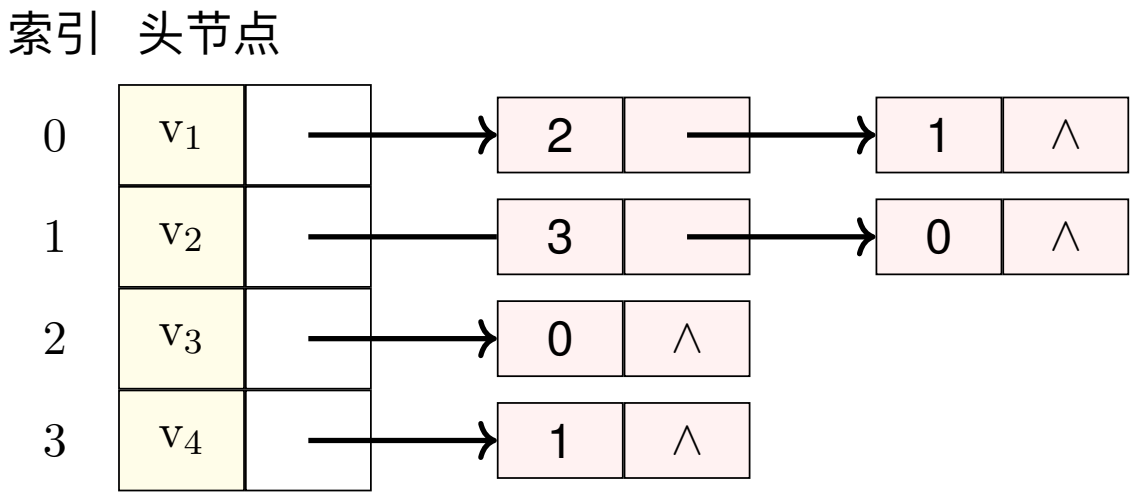


$$
\begin{array}{c}
 \\
v_1 \\
v_2 \\
v_3 \\
v_4
\end{array}
\begin{array}{c}
\begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\
\begin{pmatrix} \infty & 8 & 3 & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 1 \\ 5 & \infty & \infty & \infty \end{pmatrix}
\end{array}
$$

- 有些图的边带有权重 (常用来表示成本、距离、时间等), 这样的图称为：网。
- 网的邻接矩阵表达权重, 没有边的顶点之间的权重默认为 ∞
- 邻接矩阵表示方法非常直观、简单, 但是会有什么问题?
- 现实中的图经常对应稀疏矩阵, 在这样情形下会有很大空间浪费.

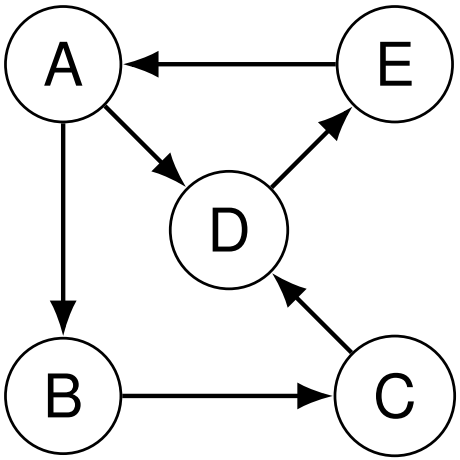
邻接表 (Adjacency List) – 无向图



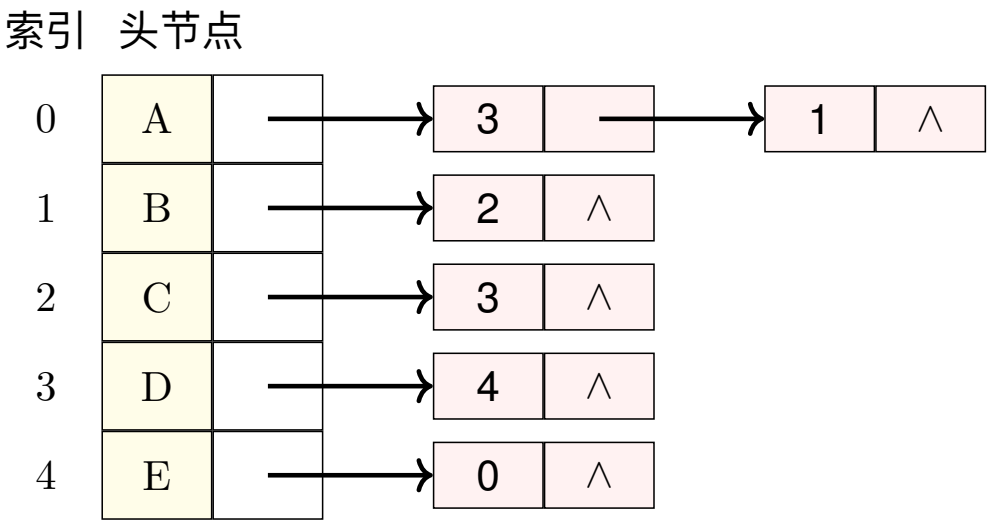


- 无向图的邻接表: 同一个顶点发出的边链接在同一个边链表中, 便于确定顶点的度
- 需要 n 个头结点, 2e 个表结点

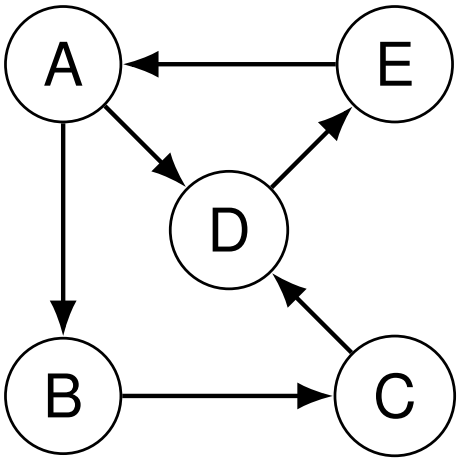
邻接表--有向图



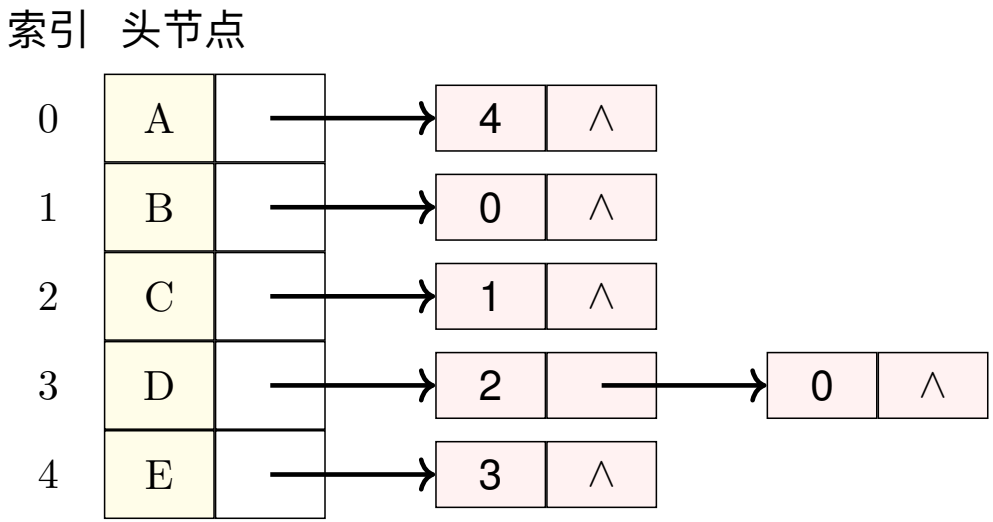
邻接表, 便于确定节点出度



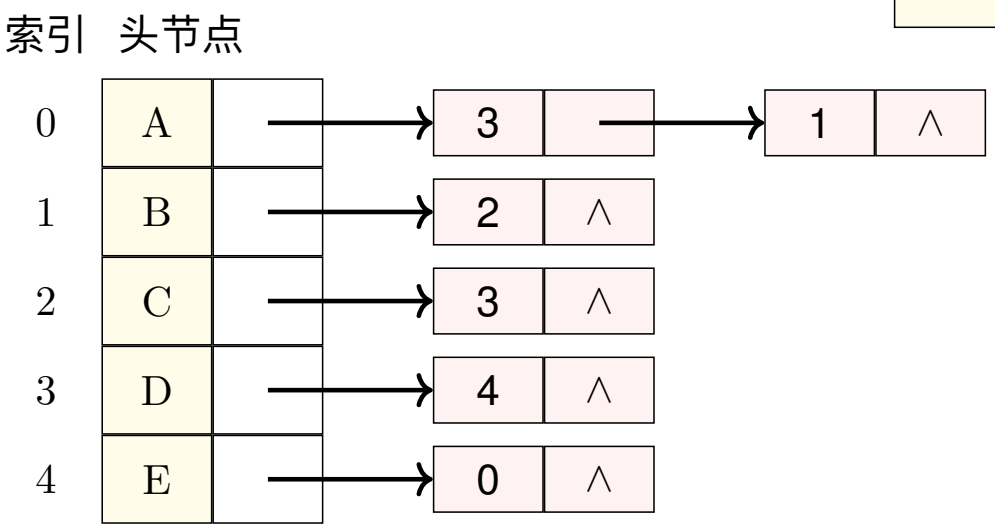
邻接表--有向图



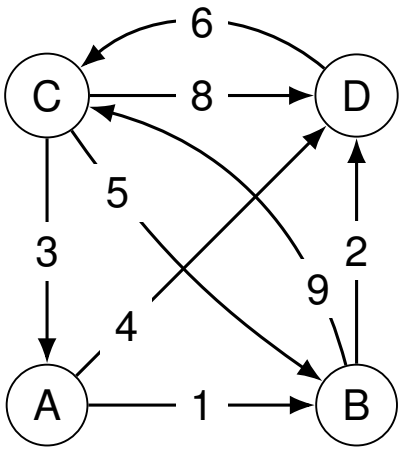
逆邻接表, 便于确定节点入度

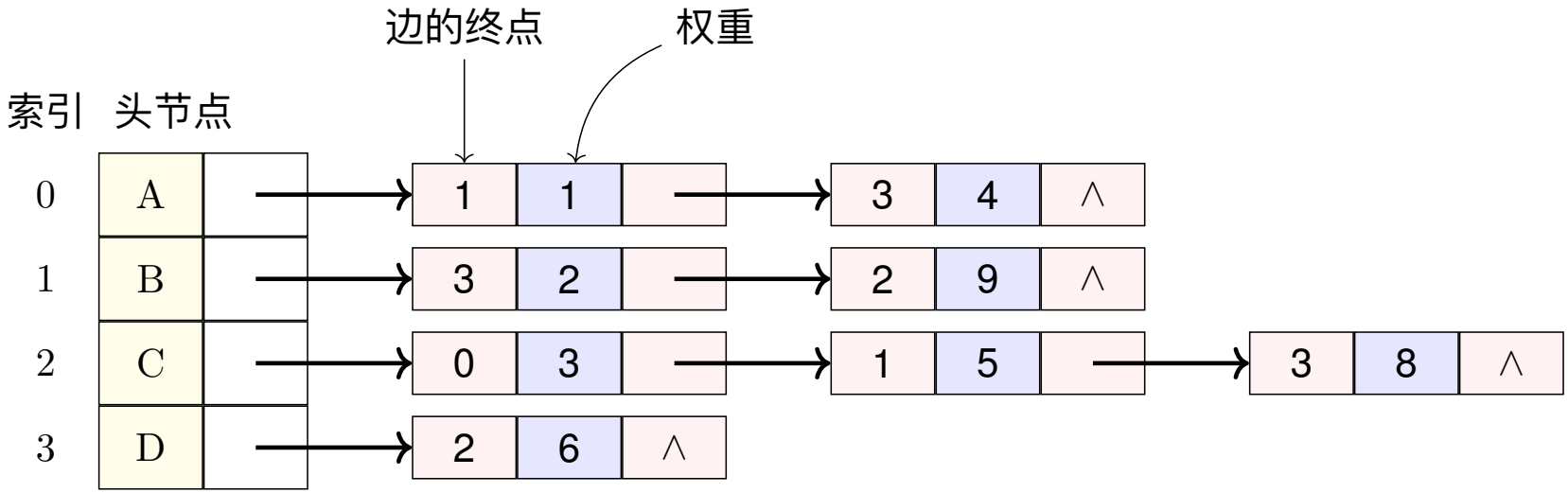


邻接表, 便于确定节点出度



邻接表--权重处理





练习

1. 请写出数组存储和邻接表的类型定义
2. 请在如下方面对比数组表示法和邻接表示法
   - 存储表示是否唯一
   - 空间复杂度
   - 操作 a: 求顶点 v_i 的度
   - 操作 b: 判定 (v_i, v_j) 是否是图的一条边
   - 操作 c: 通过遍历求边的数目

邻接表表示 I

```
class VertexNode {
  String data;
  EdgeNode firstAdj = null;

  public VertexNode(String data) {
    this.data = data;
  }
}

class EdgeNode {
  int adjVertexNode;
  EdgeNode nextAdj = null;

  public EdgeNode(int vertexIdx) {
    this.adjVertexNode = vertexIdx;
  }
```

邻接表表示 II

```
    public EdgeNode(int vertexIdx, EdgeNode nextAdj) {
      this.adjVertexNode = vertexIdx;
      this.nextAdj = nextAdj;
    }
  }

  public class Graph {
    VertexNode[] vertices;

    public void init() {
      this.vertices = new VertexNode[]{
        new VertexNode("v1"),
        new VertexNode("v2"),
        new VertexNode("v3"),
        new VertexNode("v4"),
        new VertexNode("v5"),
        new VertexNode("v6"),
```

邻接表表示 III

```
            new VertexNode("v7"),
            new VertexNode("v8")
        };
        vertices[0].firstAdj = new EdgeNode(1, new EdgeNode(2));
        vertices[1].firstAdj = new EdgeNode(0, new EdgeNode(3, new EdgeNoc
        vertices[2].firstAdj = new EdgeNode(0, new EdgeNode(5, new EdgeNoc
        vertices[3].firstAdj = new EdgeNode(1, new EdgeNode(7));
        vertices[4].firstAdj = new EdgeNode(1, new EdgeNode(7));
        vertices[5].firstAdj = new EdgeNode(2, new EdgeNode(6));
        vertices[6].firstAdj = new EdgeNode(2, new EdgeNode(5));
        vertices[7].firstAdj = new EdgeNode(3, new EdgeNode(4));
    }
}
```

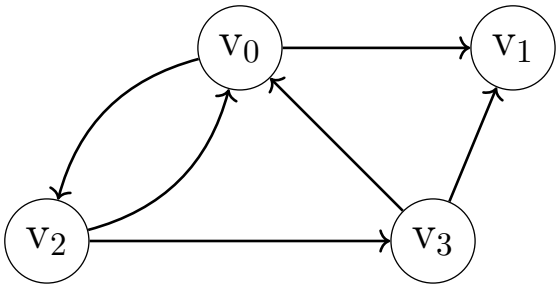
比较

	数组表示法	邻接表法
表示结果	唯一	不唯一
空间复杂度	$O(n^2)$ (适用于稠密图)	$O(n+e)$ (适用于稀疏图)
无向图求顶点 v_i 的度	第 i 行 (或第 i 列) 上非零元素的个数	第 i 个边表中的结点个数
有向图求顶点 v_i 的度	第 i 行上非零元素的个数是 v_i 出度, 第 i 列上非零元素的个数是 v_i 的入度	第 i 个边表上的结点个数, 求入度还需遍历各顶点的边表。逆邻接表则相反
判定 (v_i, v_j) 是否是图的一条边	看矩阵中的 i 行 j 列是否为 0	扫描第 i 个边表
求边的数目	检测整个矩阵中的非零元所耗费的时间是 $O(N^2)$	对每个边表的结点个数计数所耗费的时间是 $O(e+n)$

思考

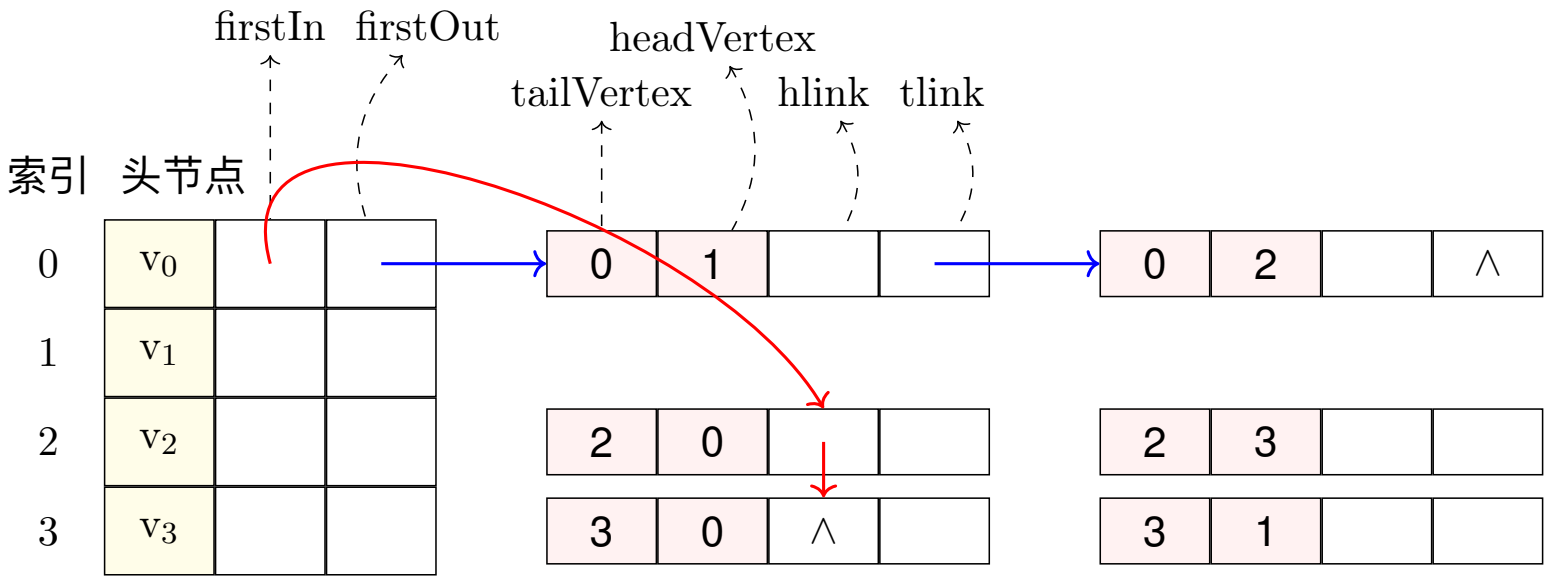
怎么把邻接表和逆邻接表相结合, 同时表示出来?

有向图的十字链表 (Orthogonal List)



将邻接表、逆邻接表结合起来.

- hlink: 指向弧头相同的下一条弧
- tlink: 指向弧尾相同的下一条弧

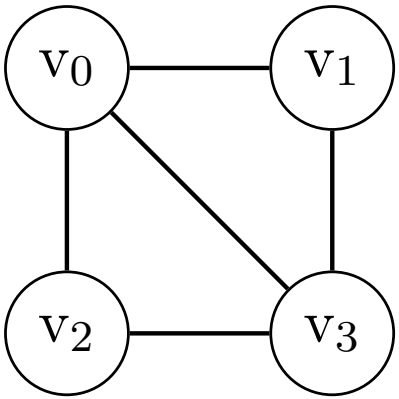


有向图的十字链表

```
class VertexNode {
  String data;
  ArcBox firstIn;
  ArcBox firstOut;
}

class ArcBox {
  int headVertex, tailVertex;
  ArcBox hlink;
  ArcBox tlink;
  String data;
}
```

```
class OLGraph {
  List<VertexNode> xlist
  int vertexNum, arcNum;
}
```

无向图的多重邻接表

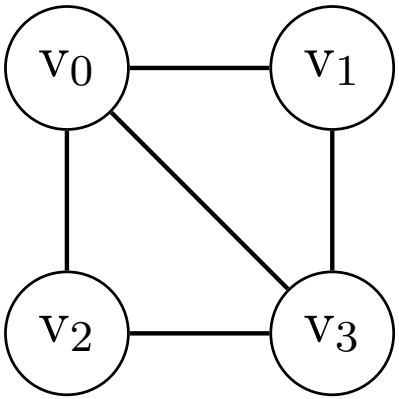


- 无向图的应用中，关注的重点是顶点，那么邻接表是不错的选择
- 如更关注边的操作，比如对已访问过的边做标记，删除某一条边等操作，就意味着需要找到这条边的两个边表结点进行操作。

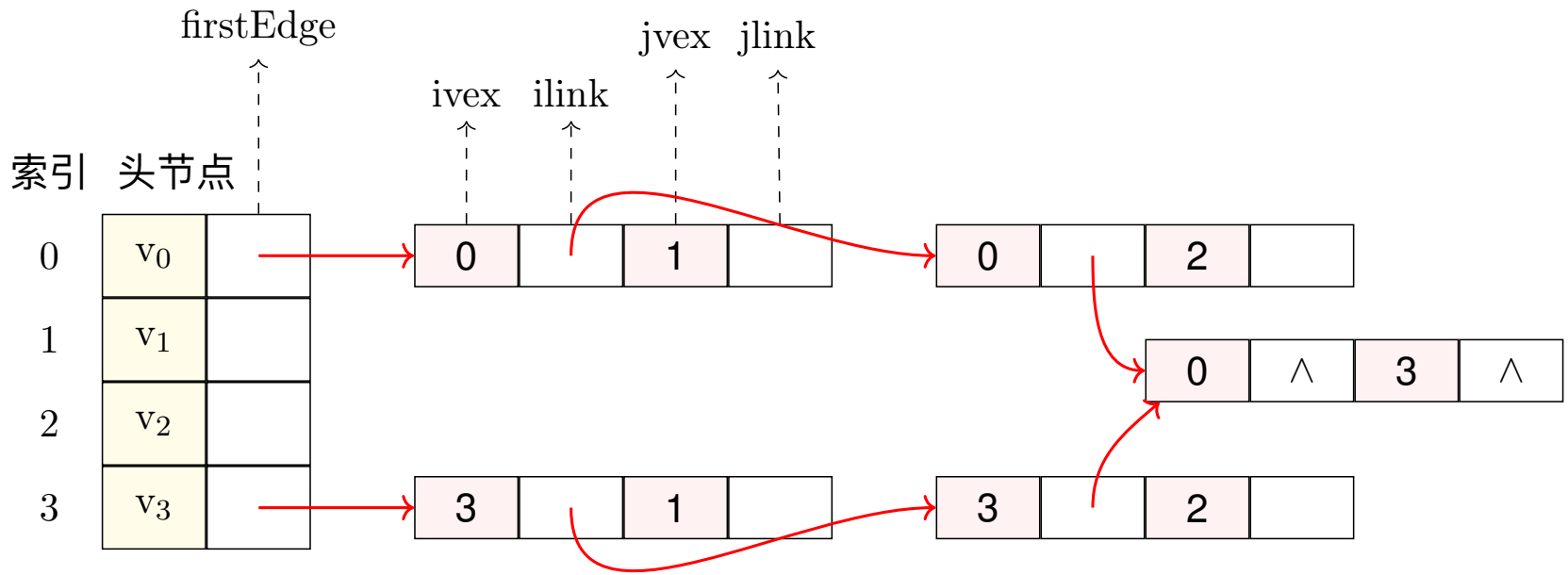
ivex	ilink	jvex	jlink

- ivex, jvex: 某条边依附的两个顶点
- ilink: 指向依附顶点 ivex 的下一条边
- jlink: 指向依附顶点 jvex 的下一条边

无向图的多重邻接表



- 无向图的应用中，关注的重点是顶点，那么邻接表是不错的选择
- 如更关注边的操作，比如对已访问过的边做标记，删除某一条边等操作，就意味着需要找到这条边的两个边表结点进行操作。



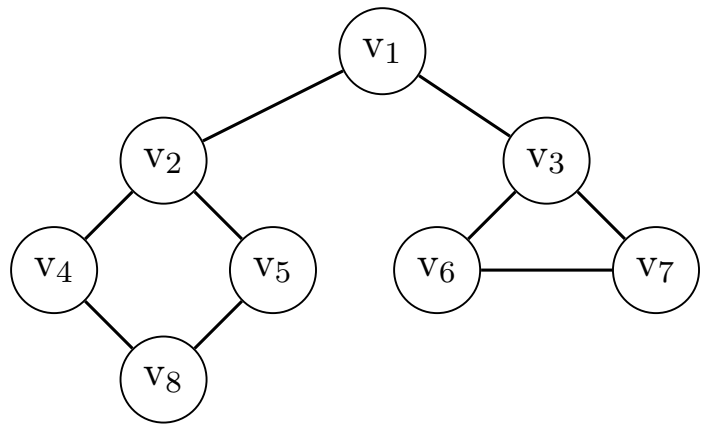
图的遍历

图的遍历

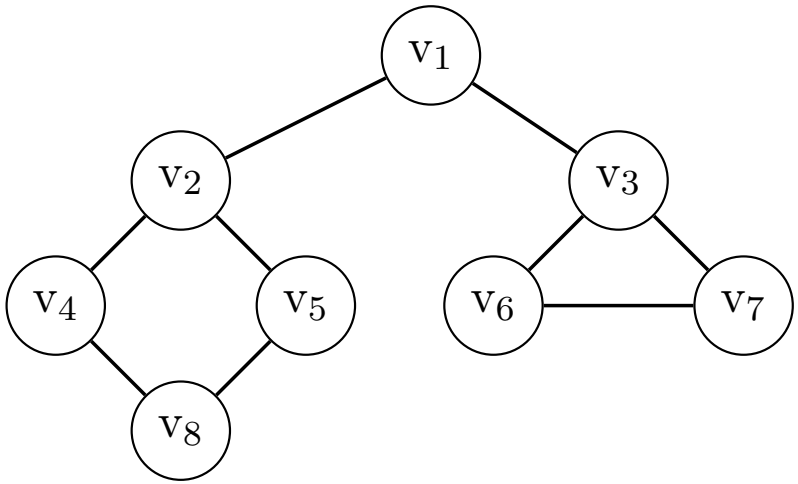
图的遍历: 从图的某顶点出发, 访问所有顶点, 且每个顶点仅被访问一次。

无论是无向图还是有向图，都有两种遍历方式：

- 深度优先 (类似于树的先根遍历)
- 广度优先 (类似于树的层次遍历)



深度优先搜索 - Depth First Search



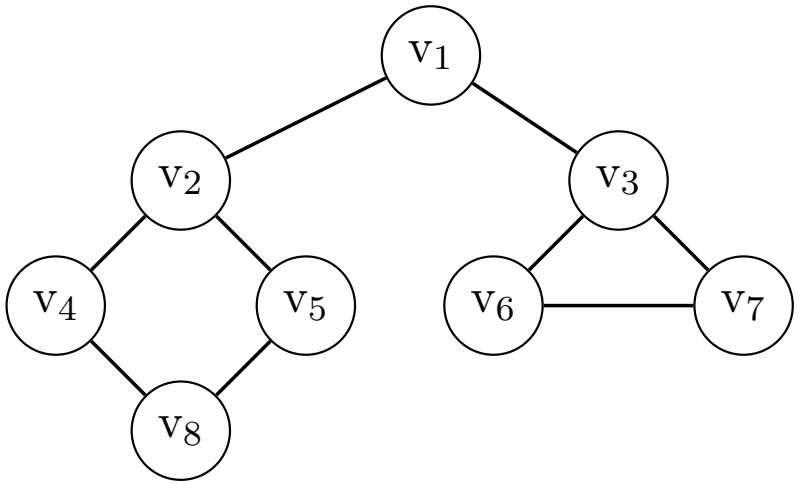
v_1	$\rightarrow v_2 \rightarrow v_3$
v_2	$\rightarrow v_1 \rightarrow v_4 \rightarrow v_5$
v_3	$\rightarrow v_1 \rightarrow v_6 \rightarrow v_7$
v_4	$\rightarrow v_2 \rightarrow v_8$
v_5	$\rightarrow v_2 \rightarrow v_8$
v_6	$\rightarrow v_3 \rightarrow v_7$
v_7	$\rightarrow v_3 \rightarrow v_6$
v_8	$\rightarrow v_4 \rightarrow v_5$

以 v_1 开始为例：

$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_8 \rightarrow v_5 \rightarrow \cdots$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
visited	1	1	0	1	1	0	0	1
stack	v_1	v_2	v_4	v_8	v_5			

深度优先搜索 - Depth First Search



v_1	$\rightarrow v_2 \rightarrow v_3$
v_2	$\rightarrow v_1 \rightarrow v_4 \rightarrow v_5$
v_3	$\rightarrow v_1 \rightarrow v_6 \rightarrow v_7$
v_4	$\rightarrow v_2 \rightarrow v_8$
v_5	$\rightarrow v_2 \rightarrow v_8$
v_6	$\rightarrow v_3 \rightarrow v_7$
v_7	$\rightarrow v_3 \rightarrow v_6$
v_8	$\rightarrow v_4 \rightarrow v_5$

以 v_1 开始为例：

$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_8 \rightarrow v_5 \rightarrow \cdots$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
visited	1	1	0	1	1	0	0	1
stack	v_1	v_2	v_4	v_8	v_5			

$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_8 \rightarrow v_5 \rightarrow v_3 \rightarrow v_6 \rightarrow v_7$

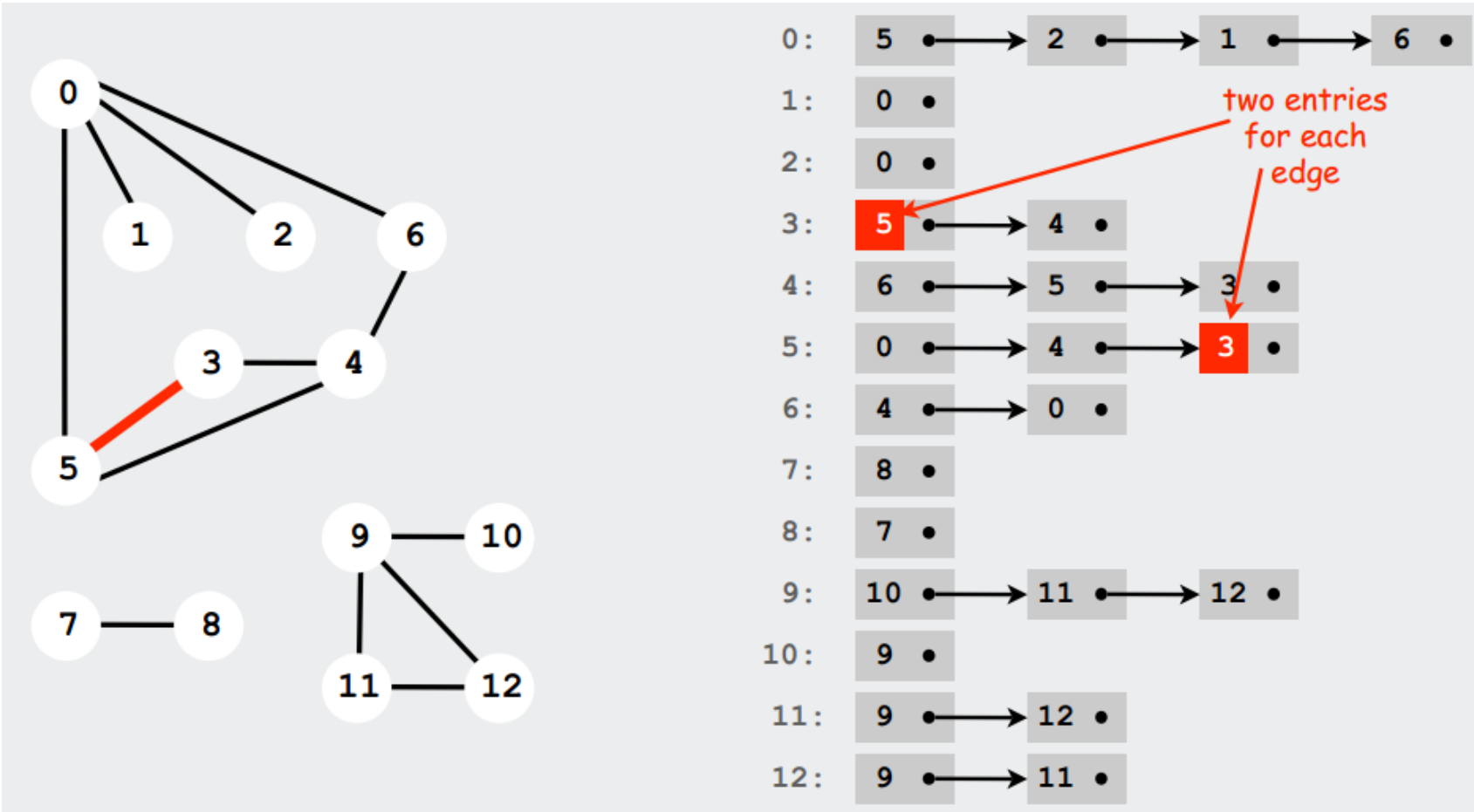
```
class VertexNode {
  String data;
  EdgeNode firstAdj = null;

  public VertexNode(String data) {
    this.data = data;
  }
}

class EdgeNode {
  int adjVertexNode;
  EdgeNode nextAdj = null;

  public EdgeNode(int vertexIdx) {
    this.adjVertexNode = vertexIdx;
  }

  public EdgeNode(int vertexIdx, EdgeNode nextAdj) {
    this.adjVertexNode = vertexIdx;
    this.nextAdj = nextAdj;
```

```
  }
}

public class Graph {
  VertexNode[] vertices;

  public void init() {
    this.vertices = new VertexNode[]{
      new VertexNode("v1"),
      new VertexNode("v2"),
      new VertexNode("v3"),
      new VertexNode("v4"),
      new VertexNode("v5"),
      new VertexNode("v6"),
      new VertexNode("v7"),
      new VertexNode("v8")
    };
    vertices[0].firstAdj = new EdgeNode(1, new EdgeNode(2));
    vertices[1].firstAdj = new EdgeNode(0, new EdgeNode(3, new EdgeNod
    vertices[2].firstAdj = new EdgeNode(0, new EdgeNode(5, new EdgeNod
```

```
    vertices[3].firstAdj = new EdgeNode(1, new EdgeNode(7));
    vertices[4].firstAdj = new EdgeNode(1, new EdgeNode(7));
    vertices[5].firstAdj = new EdgeNode(2, new EdgeNode(6));
    vertices[6].firstAdj = new EdgeNode(2, new EdgeNode(5));
    vertices[7].firstAdj = new EdgeNode(3, new EdgeNode(4));
  }

  void dfsTraverse() {
    boolean[] visited = new boolean[vertices.length];
    //for (int i = 0; i < visited.length; i++) visited[i] = false;
    for (int v = 0; v < vertices.length; v++) { //why for?
      if (!visited[v]) dfs(v, visited);
    }
  }

  void dfs(int v, boolean[] visited) {
    visited[v] = true;
    VertexNode vertex = vertices[v];
    System.out.print(vertex.data + " ");
```

```
    for (EdgeNode w = vertex.firstAdj; w != null; w = w.nextAdj) {
      if (!visited[w.adjVertexNode])
      dfs(w.adjVertexNode, visited);
    }
  }

  public static void main(String[] args) {
    Graph g = new Graph();
    g.init();
    g.dfsTraverse();
  }
}
```

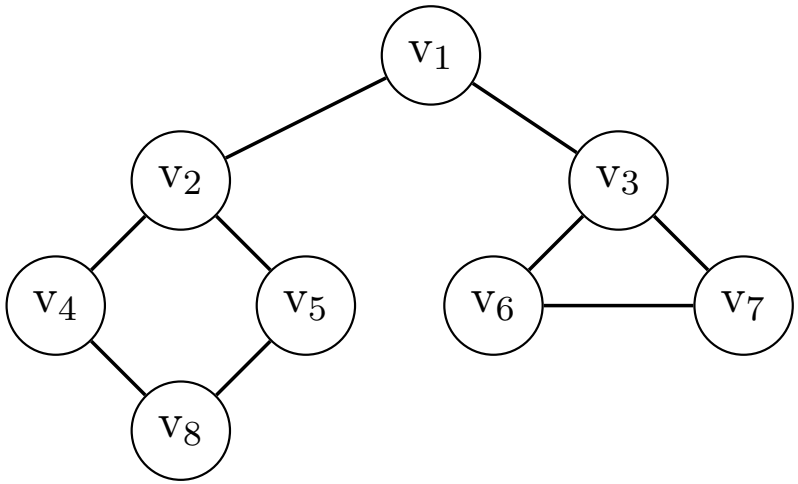
图不一定连通, 需要遍历每一个节点



DFS 算法分析

- 比较两种存储结构下的算法 (设 n 个顶点, e 条边)
  - 数组表示: 查找每个顶点的邻接点要遍历每一行, 遍历的时间复杂度为 $\mathrm{O}(\mathrm{n}^2)$
  - 邻接表表示: 虽然有 $2\mathrm{e}$ 个表结点, 但只需扫描 e 个结点即可完成遍历, 加上访问 n 个头结点的时间, 遍历的时间复杂度为 $\mathrm{O}(\mathrm{n}+\mathrm{e})$
- 结论:
  - 稠密图适于在邻接矩阵上进行深度遍历;
  - 稀疏图适于在邻接表上进行深度遍历。

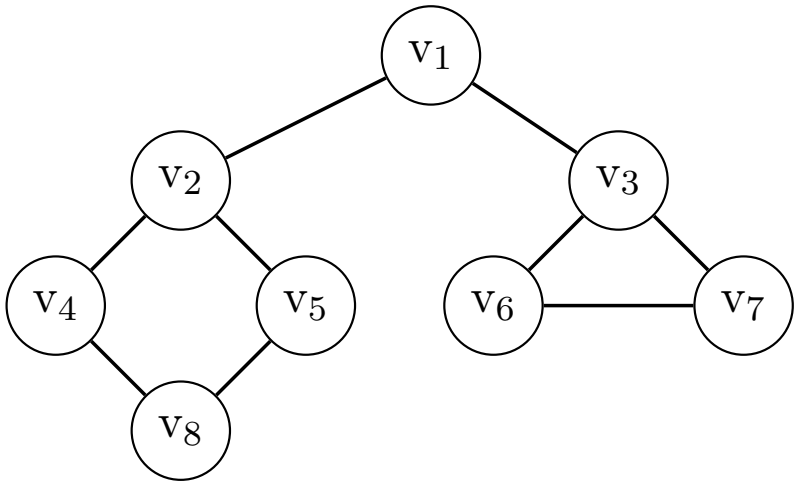
广度优先搜索 - Breadth First Search



	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
visited	1	0	0	0	0	0	0	0
queue	v_2	v_3						

v_1	$\rightarrow v_2 \rightarrow v_3$
v_2	$\rightarrow v_1 \rightarrow v_4 \rightarrow v_5$
v_3	$\rightarrow v_1 \rightarrow v_6 \rightarrow v_7$
v_4	$\rightarrow v_2 \rightarrow v_8$
v_5	$\rightarrow v_2 \rightarrow v_8$
v_6	$\rightarrow v_3 \rightarrow v_7$
v_7	$\rightarrow v_3 \rightarrow v_6$
v_8	$\rightarrow v_4 \rightarrow v_5$

广度优先搜索 - Breadth First Search



v_1	$\rightarrow v_2 \rightarrow v_3$
v_2	$\rightarrow v_1 \rightarrow v_4 \rightarrow v_5$
v_3	$\rightarrow v_1 \rightarrow v_6 \rightarrow v_7$
v_4	$\rightarrow v_2 \rightarrow v_8$
v_5	$\rightarrow v_2 \rightarrow v_8$
v_6	$\rightarrow v_3 \rightarrow v_7$
v_7	$\rightarrow v_3 \rightarrow v_6$
v_8	$\rightarrow v_4 \rightarrow v_5$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
visited	1	0	0	0	0	0	0	0
queue	v_2	v_3						

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \rightarrow v_8$$

BFS I

```
void bfs() {
  boolean[] visited = new boolean[vertices.length];
  //for (int i = 0; i < visited.length; i++) visited[i]

  Queue<Integer> Q = new LinkedList<>();
  for (int v = 0; v < vertices.length; v++) {
    if (!visited[v]) {
      visited[v] = true;
      System.out.print(vertices[v].data + " ");
      Q.add(v);

      while (!Q.isEmpty()) {
        int u = Q.poll();
```

BFS II

```
            for (EdgeNode w = vertices[u].firstAdj; w != nu
              if (!visited[w.adjVertexNode]) {
                visited[w.adjVertexNode] = true;
                System.out.print(vertices[w.adjVertexNode].
                Q.add(w.adjVertexNode);
              }
            }
          }
        }
      }
    }
```

分析以下代码的输出结果 I

```
void bfs() {
    boolean[] visited = new boolean[vertices.length];
    //for (int i = 0; i < visited.length; i++) visited[

    Queue<Integer> Q = new LinkedList<>();
    for (int v = 0; v < vertices.length; v++) {
        if (!visited[v]) {
            Q.add(v);
        }

        while (!Q.isEmpty()) {
            int u = Q.poll();
```

分析以下代码的输出结果 II

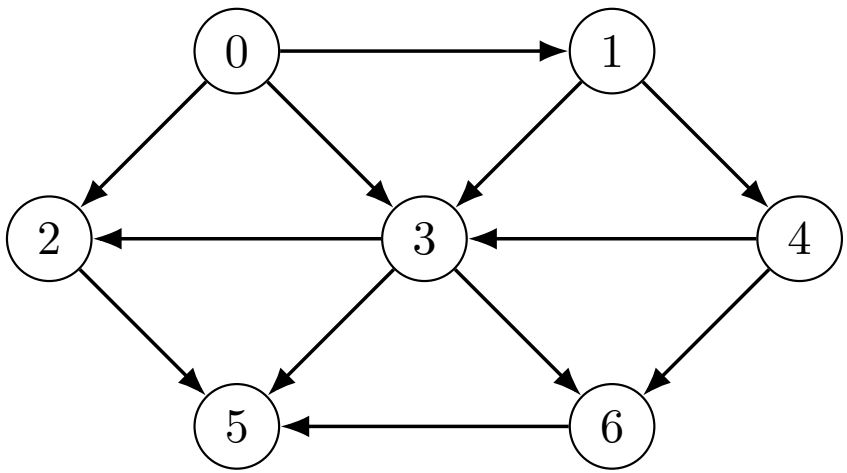
```
                visited[u] = true;
                System.out.print(vertices[u].data + " ");

                for (EdgeNode w = vertices[u].firstAdj; w !
                    if (!visited[w.adjVertexNode]) {
                        Q.add(w.adjVertexNode);
                    }
                }
            }
        }
    }
```

BFS 算法分析

- 数组表示:BFS 对于每一个被访问到的顶点, 都要循环检测矩阵中的整整一行 (n 个元素), 总的时间代价为 $\mathrm{O}(\mathrm{n}^2)$
- 邻接表表示: 时间复杂度 $\mathrm{O}(\mathrm{n}+\mathrm{e})$

作业练习

1. 请写出如下有向图的邻接矩阵, 基于该矩阵进行图的深度优先遍历;
2. 建立如下有向图的邻接表, 进行图的广度优先遍历.

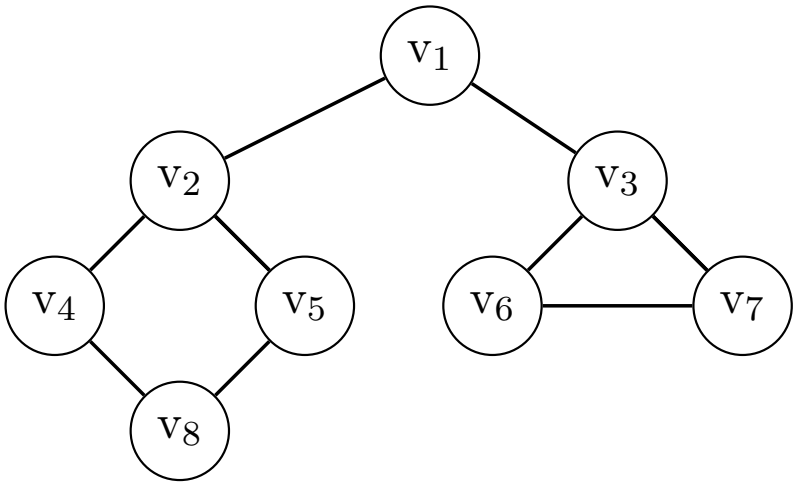


图的连通性

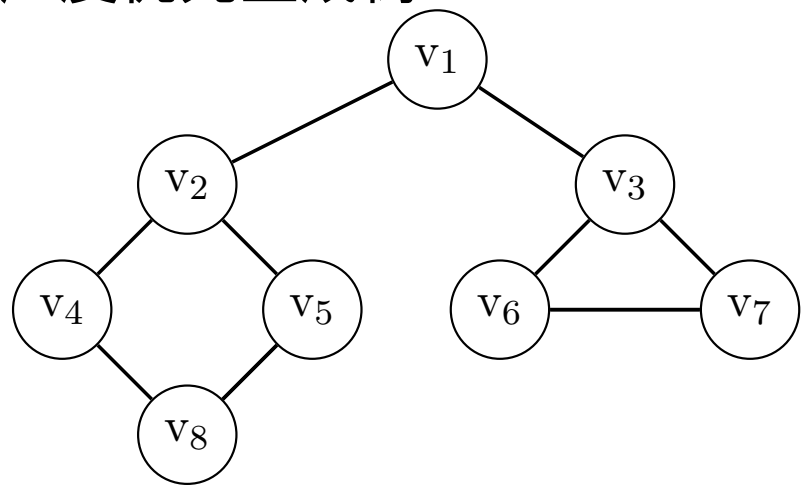
图的连通性在计算机网、通信网和电力网等方面有着重要的应用。

生成树 (Spanning tree)

深度优先生成树：



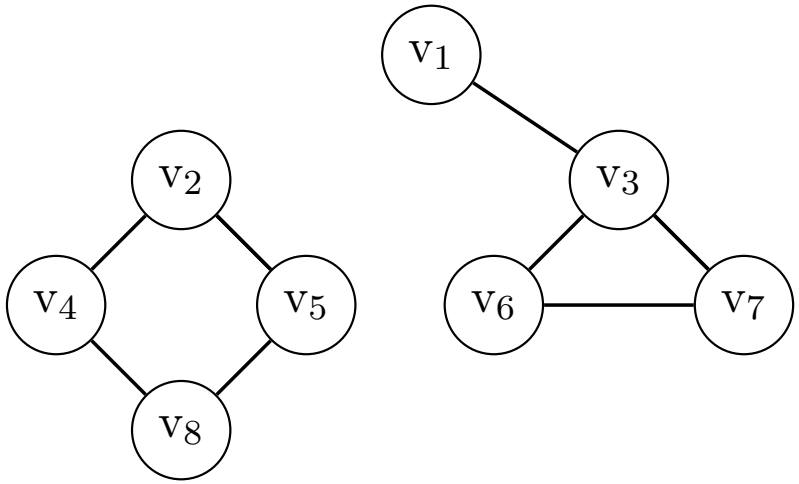
广度优先生成树：



连通图的生成树是它的极小连通子图, 有 n 个顶点和 $n-1$ 条边。

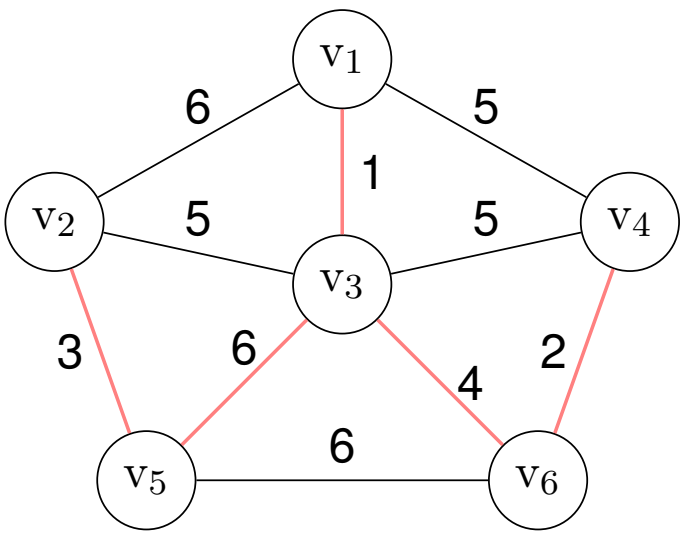
非连通图的连通分量

对于非连通图，则遍历生成森林

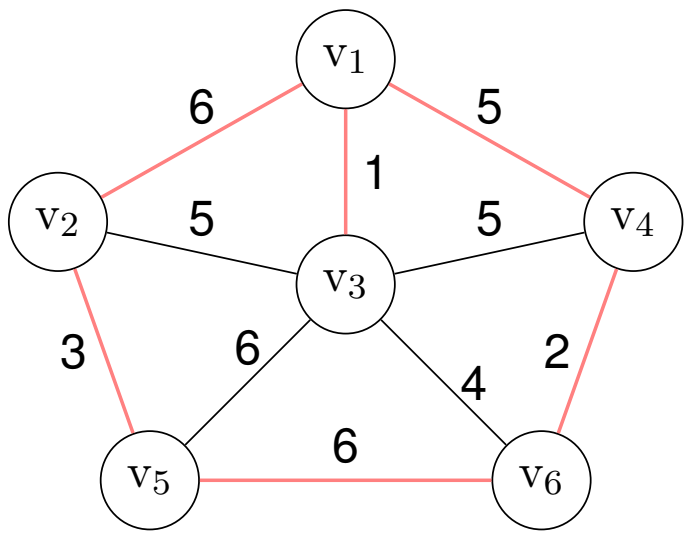


最小生成树

- 很多现实问题可以抽象成网。比如, 在 n 个城市之间建立通信网, 要求总成本最低。
- 上述问题是求连通网的最小生成树问题, 即挑选 $n-1$ 条不产生回路的最短边, 则总成本 (生成树的各边的权重之和) 达到最低。



总成本为 16



总成本为 23

最小生成树

利用最小生成树的如下性质：

假设 $G = (V, E)$ 是一个连通图,U 是顶点集 V 的一个非空子集。若 (u, v) 是一条具有最小权值 (代价) 的边, 其中 $u \in U$, $v \in V - U$, 则必存在一棵包含边 (u, v) 的最小生成树。

Prim 算法

从顶点出发：找连通部分和未连通部分之间的最小代价的一条路

Prim MST I

```
public class PrimMST {
  static int minimum(CloseEdge[] closeEdges) {
    int minValue = closeEdges[0].lowcost;
    int minVex = 0;
    for (int i = 1; i < closeEdges.length; i++) {
      int lowcost = closeEdges[i].lowcost;
      if (lowcost > 0 && (lowcost < minValue || minValue == 0)) {
        minValue = lowcost;
        minVex = i;
      }
    }
    return minVex;
  }

  static void mst() {
    CloseEdge[] closeEdges = new CloseEdge[Graph.vexnum];
    closeEdges[0] = new CloseEdge(0, 0);
```

Prim MST II

```
//初始化
for (int i = 1; i < Graph.vexnum; i++) {
  closeEdges[i] = new CloseEdge(0, Graph.arcs[0][i]);
}

//默认选中了第 0 个节点，处理剩余的 n-1 个
for (int i = 1; i < Graph.vexnum; i++) {
  int k = minimum(closeEdges);
  String fromVex = Graph.labels[closeEdges[k].adjvex];
  String toVex = Graph.labels[k];

  System.out.println(fromVex + " -> " + toVex);
  closeEdges[k].lowcost = 0;

  //处理每一个 Vertex, 看能否通过 k 让代价更低
  for (int j = 0; j < Graph.vexnum; j++) {
```

Prim MST III

```
            if (Graph.arcs[k][j] < closeEdges[j].lowcost) {
              closeEdges[j].lowcost = Graph.arcs[k][j];
              closeEdges[j].adjvex = k;
            }
          }
        }
      }

      public static void main(String[] args) {
        mst();
      }

      public static class CloseEdge {
        public int adjvex;
        public int lowcost;

        public CloseEdge(int adjvex, int lowcost) {
```

Prim MST IV

```
      this.adjvex = adjvex;
      this.lowcost = lowcost;
    }
  }


  public static class Graph {
    public static int INFINITE = 10000;

    public static int vexnum = 6;

    public static String[] labels = new String[]{"v1", "v2", "v3",
      "v4", "v5", "v6"};

    public static int[][] arcs = new int[][]{
      {0, 6, 1, 5, INFINITE, INFINITE},
      {6, 0, 5, INFINITE, 3, INFINITE},
```

Prim MST V

```
        {1, 5, 0, 5, 6, 4},
        {5, INFINITE, 5, 0, INFINITE, 2},
        {0, 3, 6, INFINITE, 0, 6},
        {INFINITE, INFINITE, 4, 2, 6, 0}
      };
    }
}
```

延伸阅读

Algorithms Course - Graph Theory Tutorial from a Google Engineer: https://www.youtube.com/watch?v=09_LlHjoEiY

本章作业

1. 最小生成树的 Prim, Kruscal 算法
2. 最短路径的 Dijstra, Floyd 算法

- 编程实现上述算法 (务必认真写注释),
- 要求显示某图的最小生成树/某两点之间的最短路径;
- 基本要求：Prim, Kruscal 可以二选一, Dijstra, Floyd 可以二选一
- 优秀要求：四种算法都实现