

数据结构

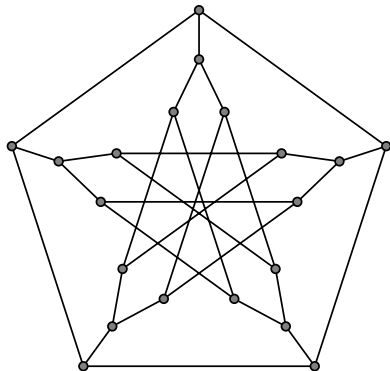
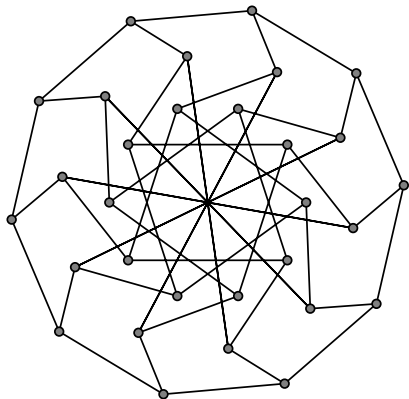
Data Structure

Xia Tian

Email: `xiat(at)ruc.edu.cn`

Renmin University of China

Graph



Content



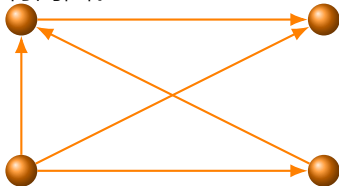
- 图的定义
- 图的存储表示
- 图的遍历
- 图的连通性



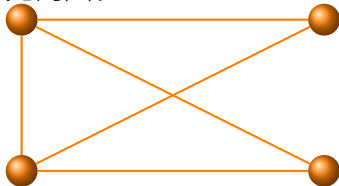
图 (Graph)

- 图 $G = (V, E)$, 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

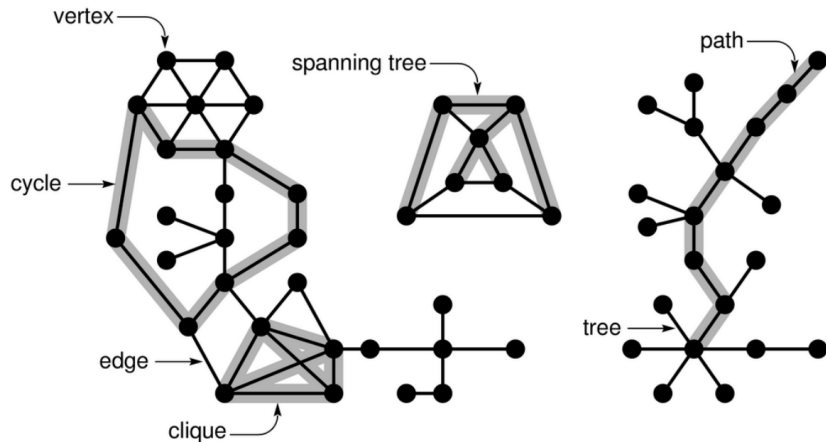
有向图:



无向图:



图的相关概念

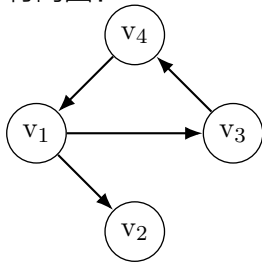




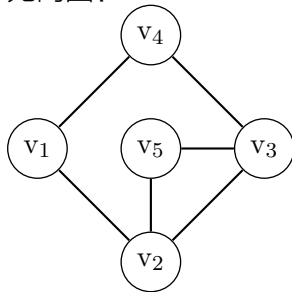
图的存储

如何表达下图的信息?

有向图:



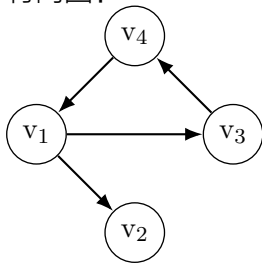
无向图:



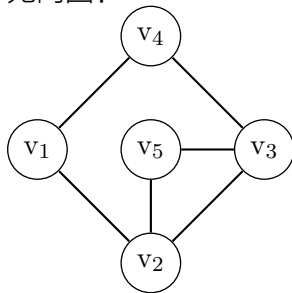
图的存储

如何表达下图的信息?

有向图:

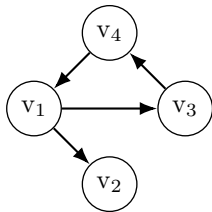


无向图:

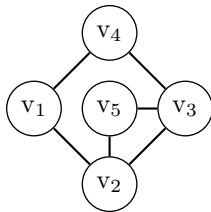


- 可用邻接矩阵表达顶点及其关系。

图的存储



	v1	v2	v3	v4
v1	0	1	1	0
v2	0	0	0	0
v3	0	0	0	1
v4	1	0	0	0

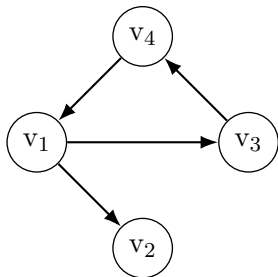


	v1	v2	v3	v4	v5
v1	0	1	0	1	0
v2	1	0	1	0	1
v3	0	1	0	1	1
v4	1	0	1	0	0
v5	0	1	1	0	0

- 根据邻接矩阵, 如何判断各顶点的度?



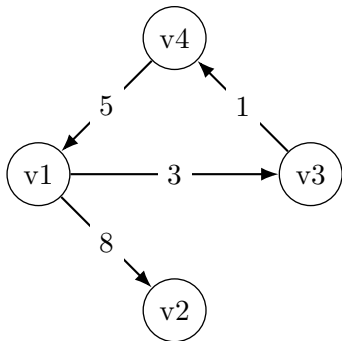
有向图的连续存储方式：邻接矩阵




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

	v_1	v_2	v_3	v_4
v_1	0	1	1	0
v_2	0	0	0	0
v_3	0	0	0	1
v_4	1	0	0	0

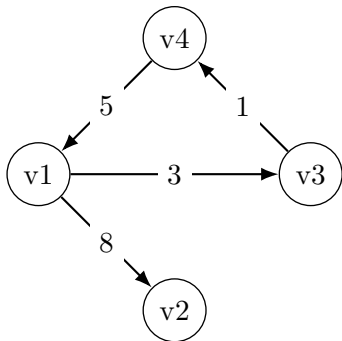
网的邻接矩阵




- 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?

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

网的邻接矩阵

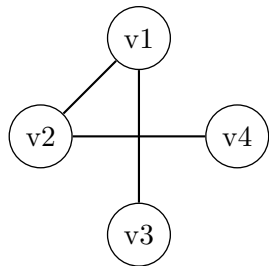


$$\begin{array}{c}
 \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}
 \begin{pmatrix}
 & v_1 & v_2 & v_3 & v_4 \\
 v_1 & \infty & 8 & 3 & \infty \\
 v_2 & \infty & \infty & \infty & \infty \\
 v_3 & \infty & \infty & \infty & 1 \\
 v_4 & 5 & \infty & \infty & \infty
 \end{pmatrix}
 \end{array}$$

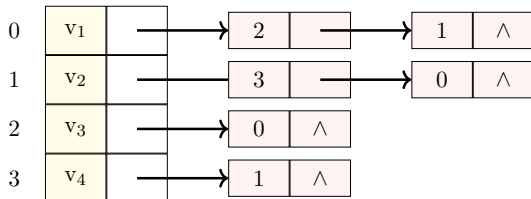
- 有些图的边带有权重 (常用来表示成本、距离、时间等), 这样的图称为: 。
- 网的邻接矩阵表达权重, 没有边的顶点之间的权重默认为 ∞
- 邻接矩阵表示方法非常直观、简单, 但是会有什么问题?
- 现实中的图经常对应稀疏矩阵, 在这样情形下会有很大空间浪费。



邻接表 (Adjacency List) – 无向图



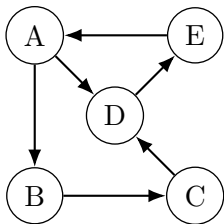
索引 头节点



- 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



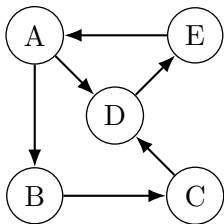
邻接表, 便于确定节点出度

索引 头节点

0	A	→	3	→	1	∧
1	B	→	2	→	∧	
2	C	→	3	→	∧	
3	D	→	4	→	∧	
4	E	→	0	→	∧	



邻接表-有向图



邻接表, 便于确定节点出度

索引 头节点

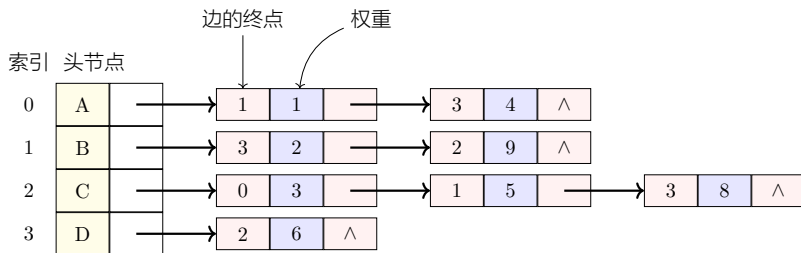
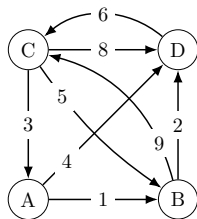
0	A	→	3	→	1	∧
1	B	→	2	∧		
2	C	→	3	∧		
3	D	→	4	∧		
4	E	→	0	∧		

逆邻接表, 便于确定节点入度

索引 头节点

0	A	→	4	∧		
1	B	→	0	∧		
2	C	→	1	∧		
3	D	→	2	→	0	∧
4	E	→	3	∧		

邻接表-权重处理





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;
```

邻接表表示 II



```
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));
```



邻接表表示 IV

```
vertices[1].firstAdj = new EdgeNode(0, new EdgeNode(3, new EdgeNode(5, new EdgeNode(7)));
vertices[2].firstAdj = new EdgeNode(0, new EdgeNode(5, new EdgeNode(7)));
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));
}
```

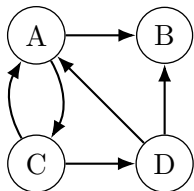
	数组表示法	邻接表法
表示结果	唯一	不唯一
空间复杂度	$O(n^2)$ (适用于稠密图)	$O(n + e)$ (适用于稀疏图)
无向图求顶点 v_i 的度	第 i 行 (或第 i 列) 上非零元素的个数	第 i 个边表中的结点个数
有向图求顶点 v_i 的度	第 i 行上非零元素的个数是 v_i 出度, 第 i 列上非零元素的个数是 v_i 的入度	第 i 个边表上的结点个数, 求入度还需遍历各顶点的边表。逆邻接表则相反
判定 (v_i, v_j) 是否是图的一条边	看矩阵中的 i 行 j 列是否为 0	扫描第 i 个边表
求边的数目	检测整个矩阵中的非零元所耗费的时间是 $O(N^2)$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



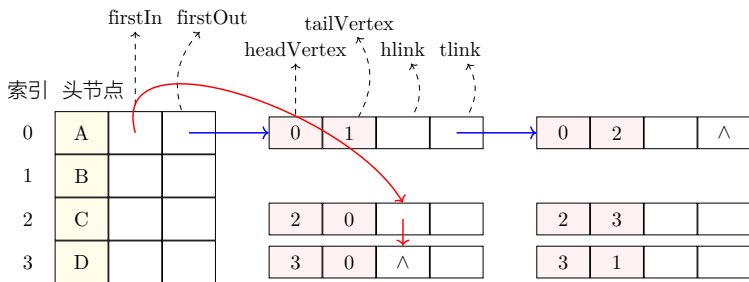
怎么把邻接表和逆邻接表相结合, 同时表示出来?



有向图的十字链表 (Orthogonal List)



将邻接表、逆邻接表结合起来.





有向图的十字链表

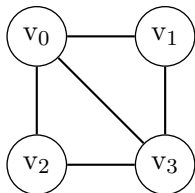
```
class VertexNode {  
    String data;  
    ArcBox firstIn;  
    ArcBox firstOut;  
}
```

```
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

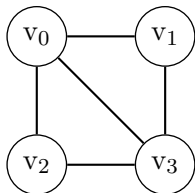


- 无向图的应用中，关注的重点是顶点，那么邻接表是不错的选择
- 如更关注边的操作，比如对已访问过的边做标记，删除某一条边等操作，就意味着需要找到这条边的两个边表结点进行操作。

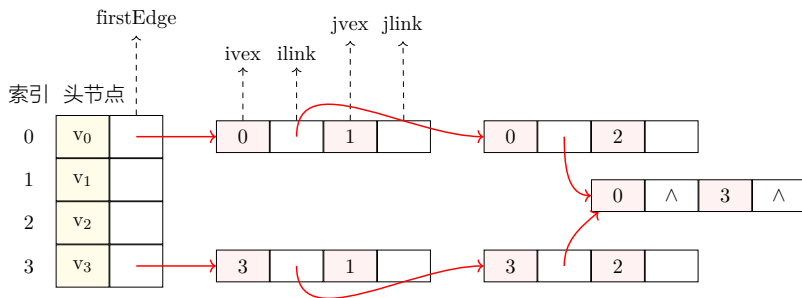
ivex	ilink	jvex	jlink
------	-------	------	-------

- 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



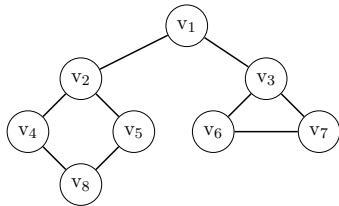


图的遍历

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。

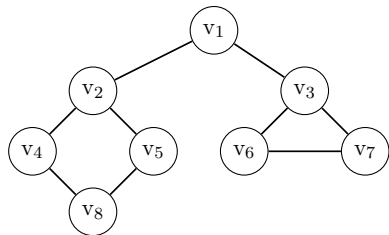
无论是无向图还是有向图, 都有两种遍历方式:

- 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



以 v_1 开始为例:

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

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

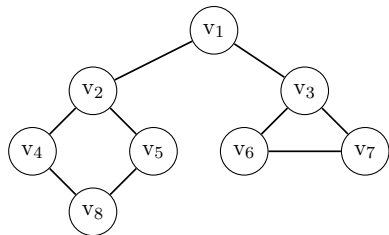
visited	1	1	0	1	1	0	0	1
---------	---	---	---	---	---	---	---	---

stack	v_1	v_2	v_4	v_8	v_5			
-------	-------	-------	-------	-------	-------	--	--	--

v_1	$\rightarrow v_2 \rightarrow v_3$
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	$\rightarrow v_4 \rightarrow v_5$



深度优先搜索 - Depth First Search



以 v_1 开始为例:

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

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

visited	1	1	0	1	1	0	0	1
---------	---	---	---	---	---	---	---	---

stack	v_1	v_2	v_4	v_8	v_5			
-------	-------	-------	-------	-------	-------	--	--	--

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

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	$\rightarrow v_4 \rightarrow v_5$

```
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 EdgeNode(4)  
vertices[2].firstAdj = new EdgeNode(0, new EdgeNode(5, new EdgeNode(6)  
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;
```

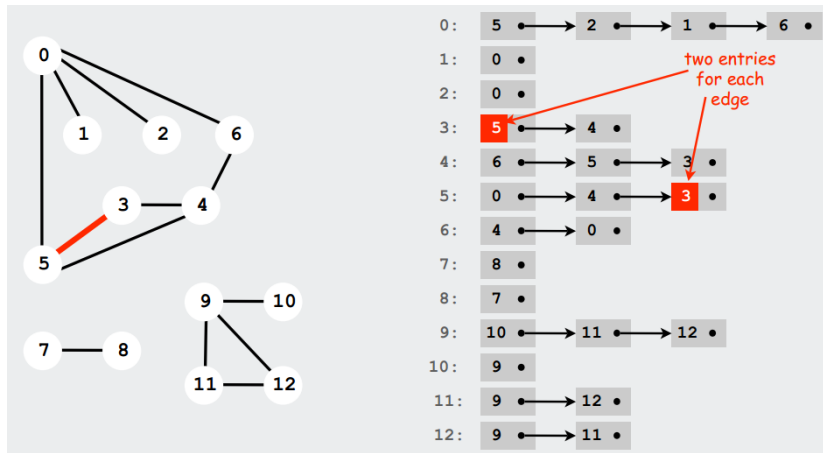
```
for (int v = 0; v < vertices.length; v++) { //why for?  
    if (!visited[v]) dfs(v, visited);  
}  
}
```

```
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

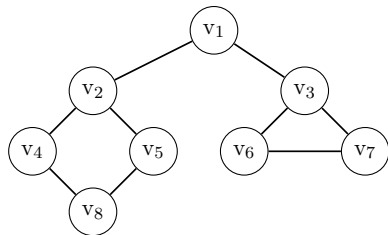




- 比较两种存储结构下的算法 (设 n 个顶点, e 条边)
 - ▶ 数组表示: 查找每个顶点的邻接点要遍历每一行, 遍历的时间复杂度为 $O(n^2)$
 - ▶ 邻接表表示: 虽然有 $2e$ 个表结点, 但只需扫描 e 个结点即可完成遍历, 加上访问 n 个头结点的时间, 遍历的时间复杂度为 $O(n + e)$
- 结论:
 - ▶ 稠密图适于在邻接矩阵上进行深度遍历;
 - ▶ 稀疏图适于在邻接表上进行深度遍历。



广度优先搜索 - Breadth First Search



v1 v2 v3 v4 v5 v6 v7 v8

visited

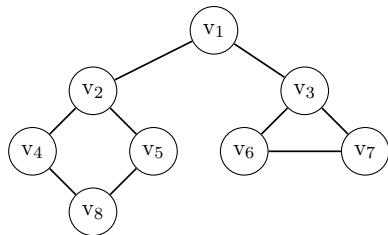
1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

queue

v2	v3						
----	----	--	--	--	--	--	--

v1	$\rightarrow v_2 \rightarrow v_3$
v2	$\rightarrow v_1 \rightarrow v_4 \rightarrow v_5$
v3	$\rightarrow v_1 \rightarrow v_6 \rightarrow v_7$
v4	$\rightarrow v_2 \rightarrow v_8$
v5	$\rightarrow v_2 \rightarrow v_8$
v6	$\rightarrow v_3 \rightarrow v_7$
v7	$\rightarrow v_3 \rightarrow v_6$
v8	$\rightarrow v_4 \rightarrow v_5$

广度优先搜索 - Breadth First Search



v1 v2 v3 v4 v5 v6 v7 v8

visited

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

queue

v2	v3						
----	----	--	--	--	--	--	--

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

v1	$\rightarrow v_2 \rightarrow v_3$
v2	$\rightarrow v_1 \rightarrow v_4 \rightarrow v_5$
v3	$\rightarrow v_1 \rightarrow v_6 \rightarrow v_7$
v4	$\rightarrow v_2 \rightarrow v_8$
v5	$\rightarrow v_2 \rightarrow v_8$
v6	$\rightarrow v_3 \rightarrow v_7$
v7	$\rightarrow v_3 \rightarrow v_6$
v8	$\rightarrow v_4 \rightarrow v_5$



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

```
    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();
```



```
for (EdgeNode w = vertices[u].firstAdj; w != null; w = w.nextAdj) {  
    if (!visited[w.adjVertexNode]) {  
        visited[w.adjVertexNode] = true;  
        System.out.print(vertices[w.adjVertexNode].data + " ");  
        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[i] = false;
```

```
    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();  
  
        visited[u] = true;
```



分析以下代码的输出结果 II

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

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

BFS 算法分析

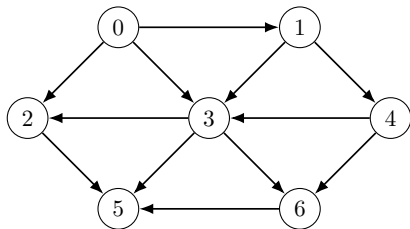


- 数组表示: BFS 对于每一个被访问到的顶点, 都要循环检测矩阵中的整整一行 (n 个元素), 总的时间代价为 $O(n^2)$
- 邻接表表示: 时间复杂度 $O(n + e)$

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

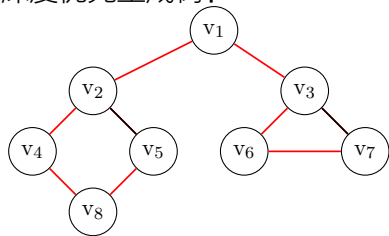


图的连通性在计算机网、通信网和电力网等方面有着重要的应用。

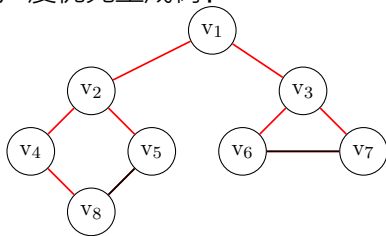


生成树 (Spanning tree)

深度优先生成树:



广度优先生成树:

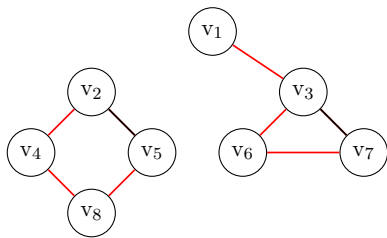


连通图的生成树是它的极小连通子图, 有 n 个顶点和 $n - 1$ 条边。

非连通图的连通分量



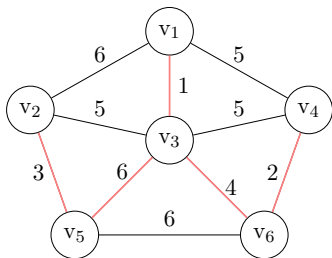
对于非连通图则遍历生成森林, 下图是深度优先遍历生成森林



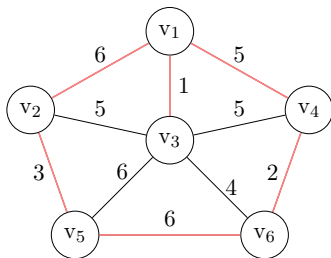


最小生成树

- 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



1. 最小生成树的 Prim, Kruscal 算法
 2. 最短路径的 Dijkstra, 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, Dijkstra, Floyd 可以二选一
 - 优秀要求: 四种算法都实现