网络空间安全数学基础（9）

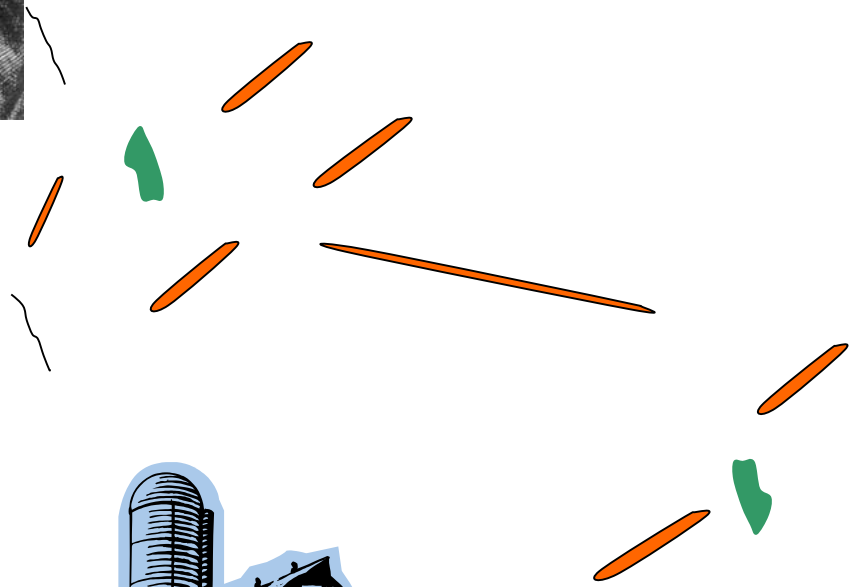
网络空间安全学院

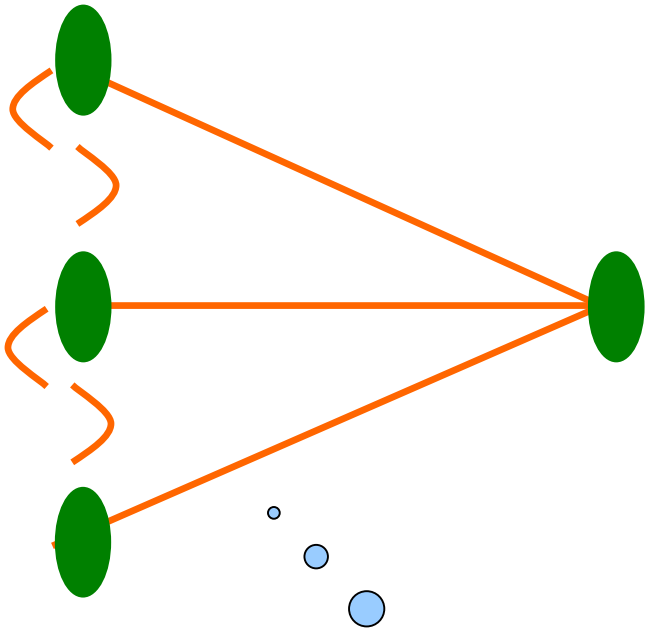
高莹

2020年 11 月 25日

哥尼斯堡七桥问题

1736年 瑞士数学家 欧拉（E. Euler) 提出“七桥问题”





是否可以
一笔画？

Euler其人

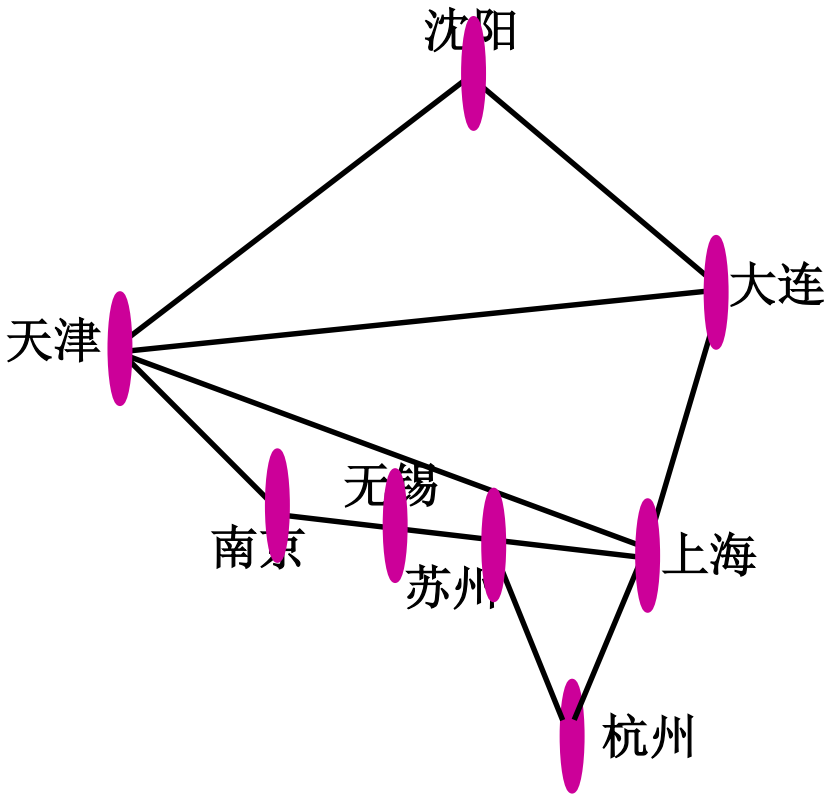

Leonhard Euler
(1707-1783)

- Leonhard Euler（1707-1783）
- 出生于瑞士的Basel（巴塞尔），在家学习数学的基础知识（身为牧师的父亲教育）。
- 后入Basel大学跟随著名的早期微积分大师之一Johann Bernoulli(1667-1748)进一步学习。
- 20岁被俄国圣彼得堡的科学院吸引为会员，在俄国工作14年，期间1735年一只眼睛失明。
- 1741年应普鲁士国王说服到柏林领导普鲁士科学院。
- 1766年另一只眼睛也失明，返回俄国圣彼得堡度过余生17年。
- 多产：影响了几乎每一个数学领域。共出版530本书和论文，在他死后，圣彼得堡科学院文献汇编继续发表他留下的手稿长达47年。
- 多子：十分喜欢孩子，曾有过13个孩子，只有5个活过了童年。一边在膝盖上照看孩子，一面做很难的计算。具有在大声干扰中做数学计算的能力以及具有比同时代人快得多的计算能力。

图论模型

例 1　你想做一次乘火车或轮船的旅行，从沈阳出发，途径大连，天津，南京，上海，杭州，苏州，无锡等城市，最后回到沈阳。那么，应该如何设计旅游线路，才能够经过每个城市一次，而且仅仅一次？

方法：点表示要去的城市，两城市间可以到达，用点之间的连线描述。



由图上可以看出旅游路线是：

沈阳→天津→南京→无锡→苏州→杭州→上海→大连→沈阳

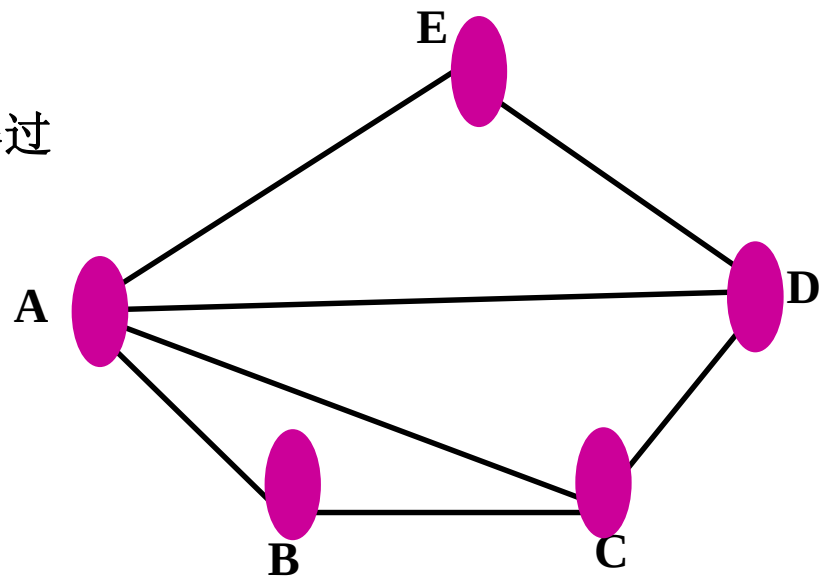
图论模型

例2　有A、B、C、D、E 五个球队举行比赛，已知A 队与其它各队都比赛过一次；B 队和A、C 队比赛过；C 队和A、B、D 队赛过； D 队和A、C、E 队比赛过；E 队和A、D 队比赛过。

那么：这种比赛关系就可以用图来表示

点：表示球队

点与点之间的连线：表示两球队间比赛过



图论模型

例3　五个球队之间的比赛结果是：A队胜了B、D、E队； B队胜了C队； C队胜了A、D队； D队没有胜过；E队胜了D队；

那么，这种胜负关系该如何用图来描述呢？

——用点与点之间带箭头的线描述“胜负关系”关系

从图中可以看出各球队之间比赛情况：

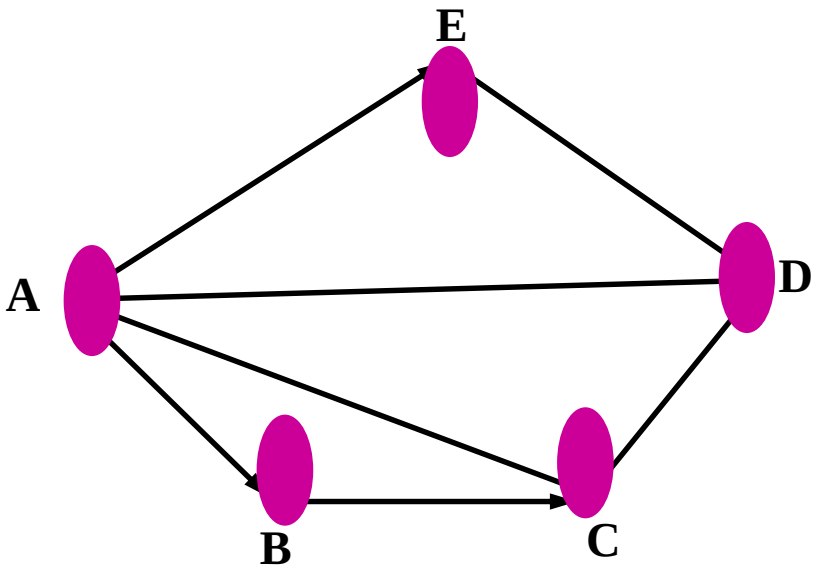
A队三胜一负

B队一胜一负

C队两胜一负

D队三战三负

E队一胜一负



图论的目的

- 将实际生活中的事物分析转化为图论问题
  - 怎么从实际生活中提炼出问题
  - 怎样通过编程求解问题

Outline

- 图的基本概念
- 图的基本算法
  - 图的遍历
    - **Breadth-first search**（广度优先搜索）
    - **depth-first search** （深度优先搜索）
  - **Minimum spanning trees (最小生成树)**
  - **Shortest Paths**（最短路径）
    - **Single-source shortest paths (单源点最短路径)**
    - **All-pairs shortest paths (所有结点间最短路径)**
  - **Max flow(最大流)**

Outline

- 图的基本概念
- 图的基本算法
  - 图的遍历
    - **Breadth-first search**（广度优先搜索）
    - **depth-first search** （深度优先搜索）
  - **Minimum spanning trees (最小生成树)**
  - **Shortest Paths**（最短路径）
    - **Single-source shortest paths (单源点最短路径)**
    - **All-pairs shortest paths (所有结点间最短路径)**
  - **Max flow(最大流)**

Graph (图)

- 图$G = (V, E)$ 由称之为结点 V 和边 E 的两个集合组成

Undirected graph
无向图

Directed graph
有向图

图的基本术语

- 1、端点和邻接点
- 在一个无向图中，若存在一条边(v_i, v_j)，则称v_i, vj为此边的两个端点，并称它们互为邻接点（adjacent），即v_i是v_j的一个邻接点，v_j也是v_i的一个邻接点。
- 在一个有向图中，若存在一条边$<v_i, v_j>$，则称此边是顶点v_i的一条出边（outedge），顶点v_j的一条入边（inedge）； 称v_i为此边的起始端点，简称起点或始点，v_j为此边的终止端点，简称终点；称v_i和v_j互为邻接点，并称v_j是v_i的出边邻接点，v_i是v_j的入边邻接点。

图的基本术语

- 2、顶点的度、入度、出度
- 无向图顶点 v 的度（degree）定义为以该顶点为一个端点的边的数目，简单地说，就是该顶点的边的数目，记为$D(v)$。
- 有向图中顶点v的度有入度和出度之分，入度（indegree）是该顶点的入边的数目，记为$ID(v)$；出度（outdegree）是该顶点的出边的数目，记为$OD(v)$顶点v的度等于它的入度和出度之和，即$D(v) = ID(v) + OD(v)$。

图的基本术语

- **3、完全图、稠密图、稀疏图**
- 若无向图中的每两个顶点之间都存在着一条边，有向图中的每两个顶点之间都存在着方向相反的两条边，则称此图为完全图。显然，若完全图是无向的，则图中包含有 $n(n-1)/2$ 条边, 若完全图是有向的，则图中包含有$n(n-1)$条边。当一个图接近完全图时，则可称为稠密图，相反地，当一个图有较少的边数（即$e << n(n-1)$）时，则可称为稀疏图。

图的基本术语

- **4、 子图**
- **设有两个图$G = (V, E)$和$G' = (V', E')$, 若V'是V的子集，且E'是E的子集，则称G'是G的子图。**

图的基本术语

- **5、路径和回路**
- 在一个图G中，从顶点v到顶点v'的一条路径（path）是一个顶点序列$v_{i0}, vi_1, vi_2, \ldots, vim$,其中$v = vi_0, v' = vim$,
- 若此图是无向图，则$(vi_{j_1}, v_{ij}) \in E(G), (1 \le j \le m)$;
- 若此图是有向图，则$< vi_{j_1}, v_{ij} > \in E(G), (1 \le j \le m)$。
- 路径长度是指该路径上经过的边的数目。
- 若一条路径上除了前后端点可以相同外，所有顶点均不同，则称此路径为简单路径。
- 若一条路径上的前后两端点相同，则被称为回路或环（cycle），前后两端点相同的简单路径被称为简单回路或简单环。

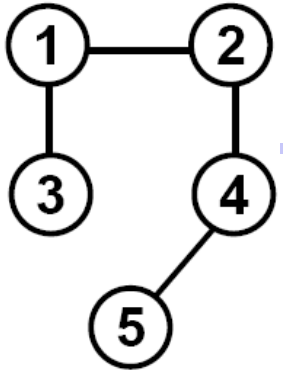
图的基本术语

- **6、连通和连通分量**
- 在无向图G中，若从顶点v_i到顶点v_j有路径，则称vi和vj是连通的。若图 G 中任意两个顶点都连通，则称 G 为连通图，否则称为非连通图。无向图G的极大连通子图称为G的连通分量。显然，任何连通图的连通分量只有一个，即本身，而非连通图有多个连通分量。

图的基本术语

- **7、权和网**
- 在一个图中．每条边可以标上具有某种含义的数值，此数值称为该边的权（**weight**）。
- 例如，对于一个反映城市交通线路的图，边上的权可表示该条线路的长度或等级；对于一个反映电子线路的图，边上的权可表示两端点间的电阻、电流或电压；对于一个反映零件装配的图，边上的权可表示一个端点需要装配另一个端点的零件的数量；对于一个反映工程进度的图，边上的权可表示从前一子工程到后一子工程所需要的天数。
- 边上带有权的图称作带权图，也常称作网（**network**）。

图的表示

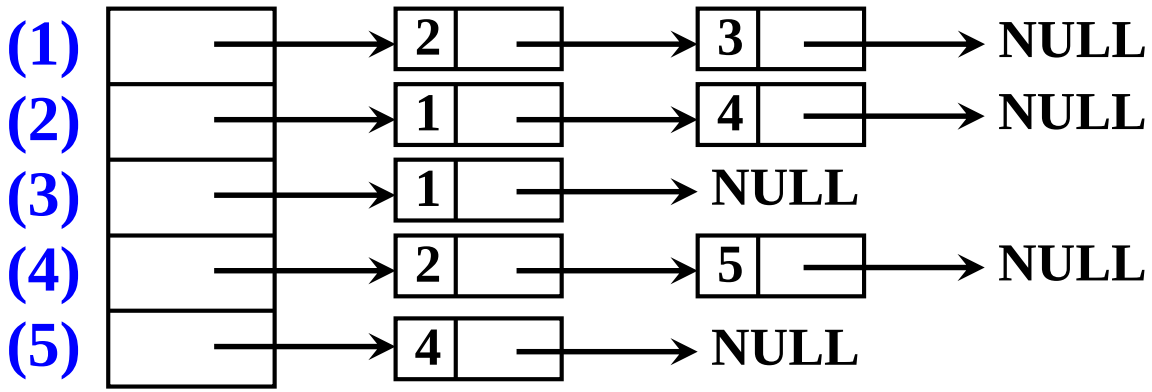


- 邻接矩阵: dense 常用于稠密图|E| ～ $|V|^2$

$$\begin{array}{c} \\ (1) \\ (2) \\ (3) \\ (4) \\ (5) \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Space: $O(n^2)$

- 邻接表: sparse 常用于稀疏图 |E|<<$|V|^2$



Space: $O(2e)$

Outline

- 图的基本概念
- 图的基本算法
  - 图的遍历
    - **Breadth-first search**（广度优先搜索）
    - **depth-first search** （深度优先搜索）
  - **Minimum spanning trees (最小生成树)**
  - **Shortest Paths**（最短路径）
    - **Single-source shortest paths (单源点最短路径)**
    - **All-pairs shortest paths (所有结点间最短路径)**
  - **Max flow(最大流)**

图的遍历

由于图结构本身的复杂性，所以图的遍历操作也较复杂，主要表现在以下四个方面：

①在图结构中，没有一个“自然”的首结点，图中任意一个顶点都可作为第一个被访问的结点。

②在非连通图中，从一个顶点出发，只能够访问它所在的连通分量上的所有顶点，因此，还需考虑如何选取下一个出发点以访问图中其余的连通分量。

③在图结构中，如果有回路存在，那么一个顶点被访问之后，有可能沿回路又回到该顶点。

④在图结构中，一个顶点可以和其它多个顶点相连，当这样的顶点访问过后，存在如何选取下一个要访问的顶点的问题。

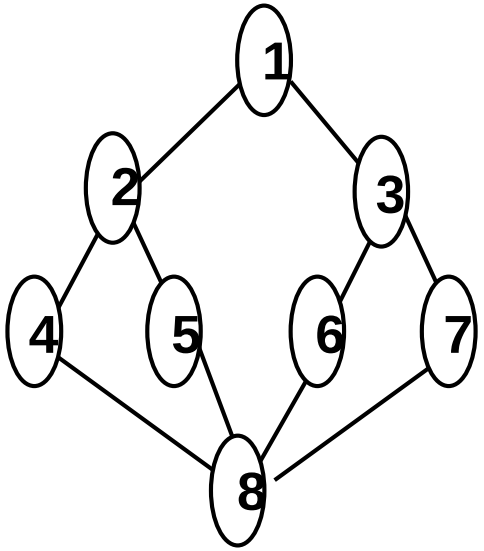
Breadth-first Search (广度优先检索)

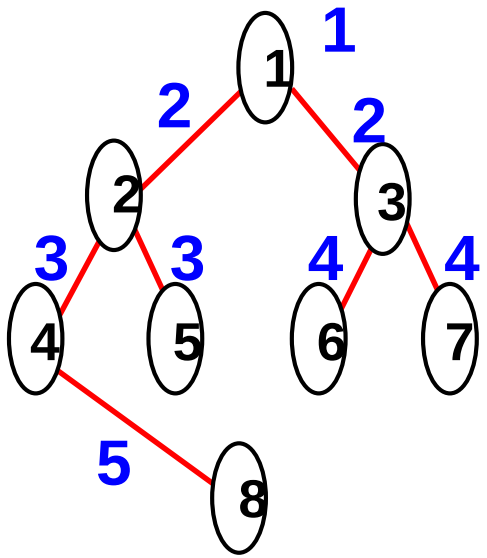
- 广度优先搜索（breadth—first search）遍历过程为：
- 首先访问初始点v_i, 并将其标记为已访问过，
- 接着访问v_i的所有未被访问过的邻接点$v_{i1}, vi_2, \cdots, v_{it}$并均标记为已访问过，
- 然后再按照$v_{i1}, vi_2, \cdots, v_{it}$的次序，访问每一个顶点的所有未被访问过的邻接点，并均标记为已访问过，
- 依次类推，直到图中所有和初始点v_i有路径相通的顶点都被访问过为止。

Breadth-first Search (广度优先检索)

- 广度优先搜索是一种分层的搜索过程，每向前走一步可能访问一批顶点。因此，广度优先搜索不是一个递归的过程。
- 为了实现逐层访问，算法中使用了一个队列，以记忆正在访问的这一层和下一层的顶点，以便于向下一层访问。
- 为避免重复访问，需要一个辅助数组visited[]，给被访问过的顶点加标记。

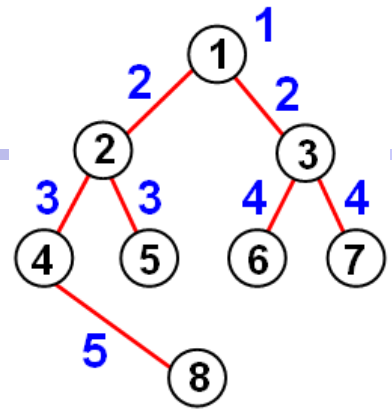
Breadth-first Search (广度优先检索)





Output: 1 2 3 4 5 6 7 8

Breadth-first Search

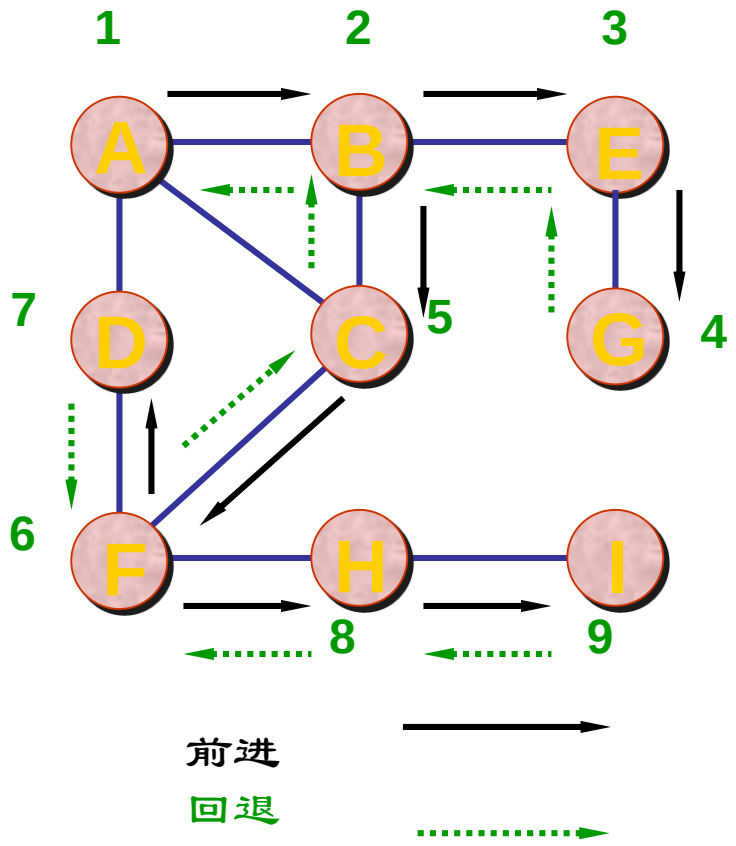



```
void BFS(int E[][], int n, int i)
// i 为开始的结点
{
    bool bVisited[n];  // 初值全部置为false
    std::queue<int> que;
    que.push(i);
    bVisited[i] = true;
    while (!que.empty())
    {
        int j = que.front(); que.pop();
        for (int k = 1; k <= n; k++)
            if (!bVisited[k] && E[j][k] ∈ E)  // 未访问且为边
                bVisited[k] = true; que.push(k);
    }
}
```

Depth-first Search (深度优先检索)

- 深度优先搜索（**depth first search**）遍历是一个递归过程.
- 可叙述为：首先访问一个顶点v_i（开始为初始点），并将其标记为已访问过，然后从v_i的一个未被访问的邻接点（无向图）或出边邻接点（有向图）出发进行深度优先搜索遍历，当v_i的所有邻接点均被访问过时，则退回到上一个顶点v_k，从v_k的另一个未被访问过的邻接点出发进行深度优先搜索遍历。

Depth-first Search (深度优先检索)



对于图的遍历，为了确保每个顶点在遍历过程中只被访问一次，需要为每个顶点建立一个访问标志visited[i]，在遍历开始之前，将它们设为 “FALSE”，一旦第i个顶点被访问，则令visited[i] 为 ″TRUE″。

讨论1：计算机如何实现DFS？ ——开辅助数组*visited*[n]！

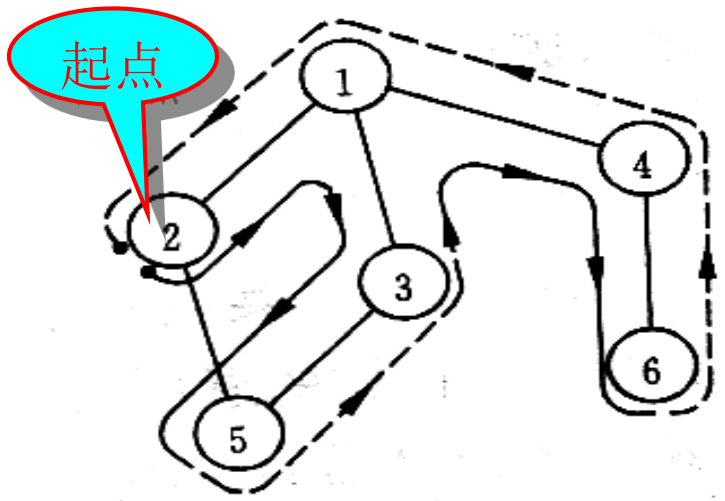
例：

邻接矩阵A

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

辅助数组 *visited* [n]

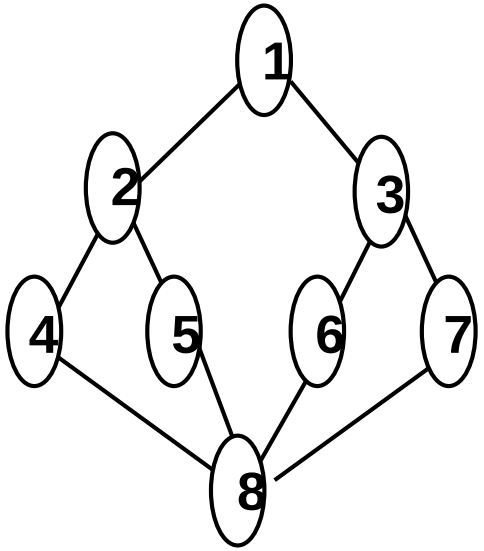
1	0	0	1	1	1	1	1
2	0	1	1	1	1	1	1
3	0	0	0	1	1	1	1
4	0	0	0	0	0	1	1
5	0	0	0	0	1	1	1
6	0	0	0	0	0	0	1

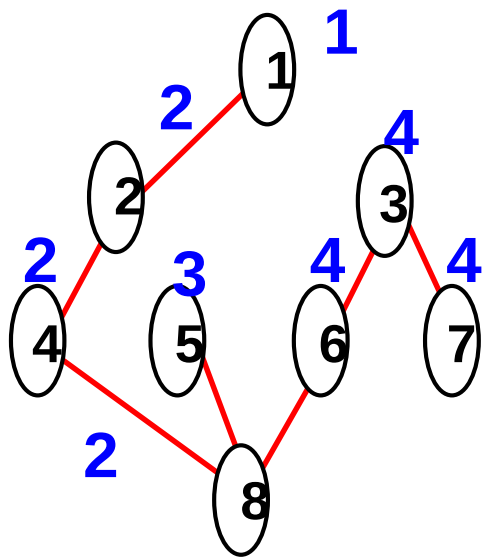


DFS 结果

v2→v1→v3→v5→v4→v6

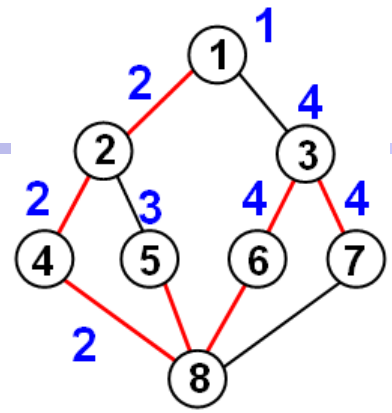
Depth-first Search (深度优先检索)





Output: 1 2 4 8 5 6 3 7

Depth-first Search




```
bool bVisited[n];  // 全局数组, 初值全部为false

void DFS(int E[][], int n, int i)
// i 为开始的结点
{
      bVisited[i] = true;

      for ( int j = 1; j <= n; j++)
      {
            if (!bVisited[j] && E[i][j] ∈ E) // 未访问且为边
                DFS(E, n, j);
      }
}
```

Outline

- 图的基本概念
- 图的基本算法
  - 图的遍历
    - Breadth-first search（广度优先搜索）
    - depth-first search （深度优先搜索）
  - Minimum spanning trees (最小生成树)
  - **Shortest Paths**（最短路径）
    - Single-source shortest paths **(**单源点最短路径**)**
    - All-pairs shortest paths **(**所有结点间最短路径**)**
  - Max flow(最大流)

Minimum Spanning Trees (最小生成树)

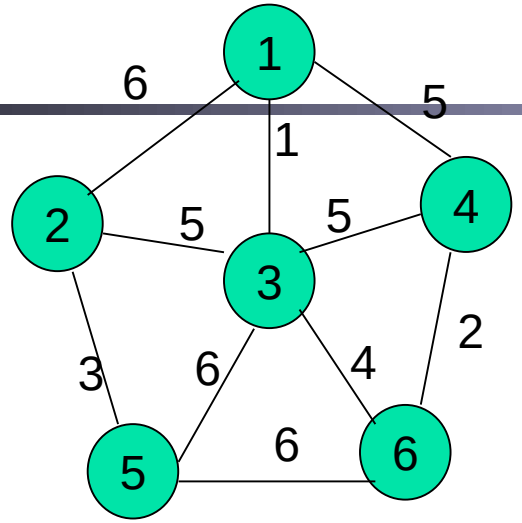
- 在设计电子线路时，常常要把数个元件的引脚连在一起，使其电位相同。要使n个引脚相互连通，可以使用$n-1$条连接线，每条连接线连接两个引脚。在各种连接方案中，通常希望找出连接线最少的接法。

图的生成树与最小生成树

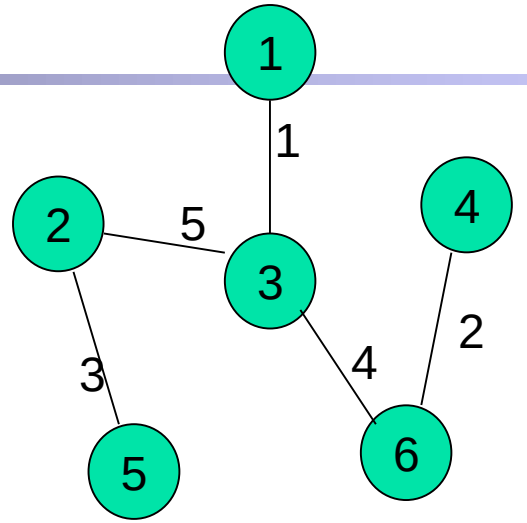
- 在一个连通图G中，如果取它的全部顶点和一部分边构成一个子图G'，即：

$$V(G') = V(G) \text{和} E(G') \subseteq E(G)$$

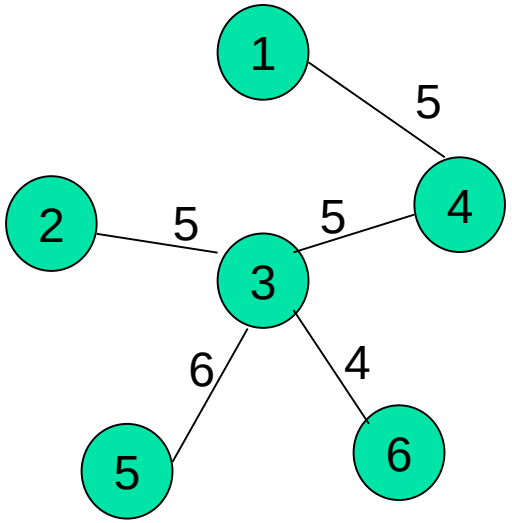
- 若边集$E(G')$中的边既将图中的所有顶点连通又不形成回路，则称
- 子图G'是原图G的一棵生成树。
- 最小生成树：给图中每个边赋一权值，所有生成树中所选择边的权值之和最小的生成树，称之为最小代价生成树，即是最小生成树。



图



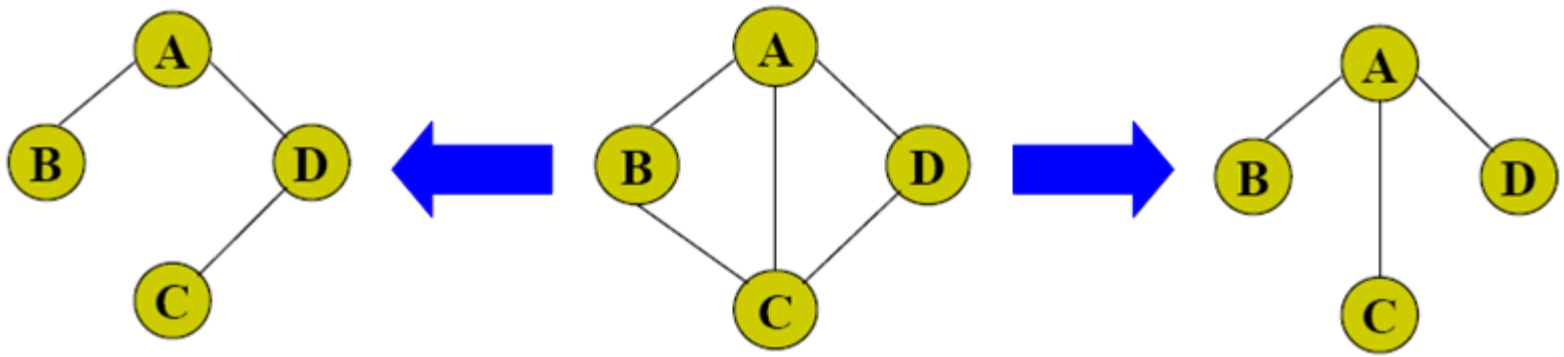
图的最小生成树



图的非最小生成树

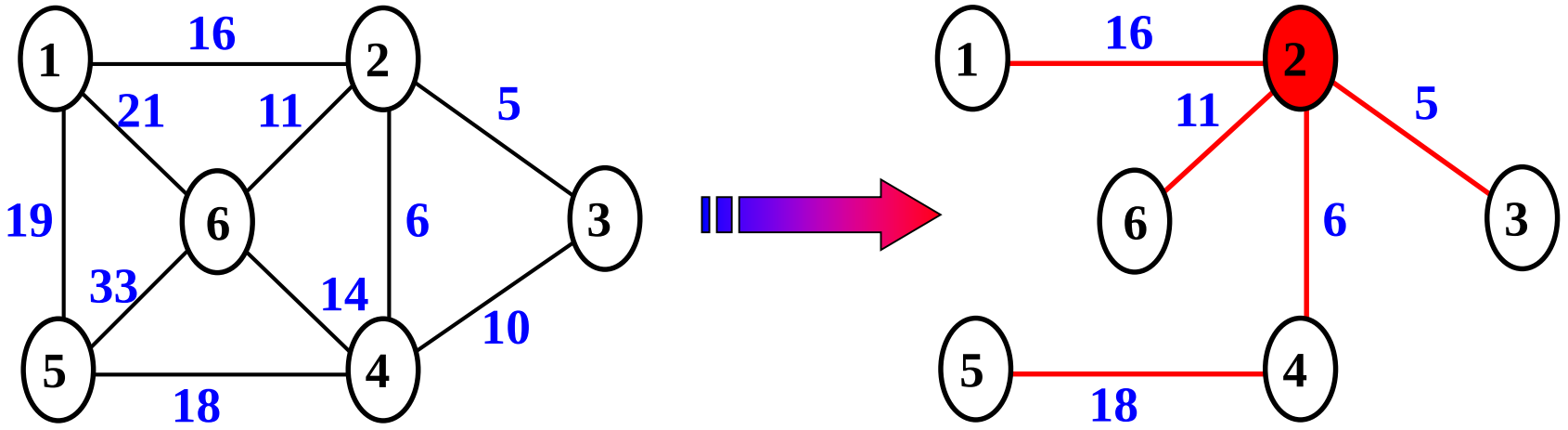
Minimum Spanning Trees (最小生成树)

- Spanning tree (生成树)
  - 设 $G = (V, E)$ 是一个无向连通图, 令 F 为追踪图时所经过的边集合, B 为未经过的边集合, 则 $S = (V, F)$ 为 G 的一棵生成树, 且 S 满足:
    - $E = F + B$
    - 从 B 中任取一边加入 S 中, 必形成环
    - 在 S 中, 任何顶点间必存在一条唯一的简单路



Minimum Spanning Trees (最小生成树)

- Minimum spanning tree problem
  - Let $G = (V, E)$ be a connected, undirected graph
  - Find a spanning tree with minimum weight



Minimum Spanning Trees (最小生成树)

- Minimum spanning tree problem
  - Let $G = (V, E)$ be a connected, undirected graph
  - Find a spanning tree with minimum weight

- Applications:
  - Wiring design
  - City traffic
  - Travel line
  - Map
  - …

Minimum Spanning Trees (最小生成树)

- Minimum spanning tree problem
  - Let $G = (V, E)$ be a connected, undirected graph
  - Find a spanning tree with minimum weight

- Algorithms by greedy method
  - Kruskal's algorithm ------ 1956
  - Prim's algorithm ------ 1957

Kruskal's Algorithm

- 集合A是一个森林，加入集合A的安全边总是图中连接两个不同连通分支的最小权边.
- 设$G=(V,E)$是一个赋权连通图，其权函数为w.
- 令$F=\Phi$.
- 当存在一条不属于F的边α, 使得$FU\{\alpha\}$中不含G中的圈时，确定这样的一条最小的边α，并将它放入F中.
- 若$|F|=n-1$, 输出图$T=(V,F)$.

算法要求：不能选择构成圈的边

Kruskal's Algorithm

Edges sorted by the weights

~~(V_1, V_2): 1~~

~~(V_3, V_5): 2~~

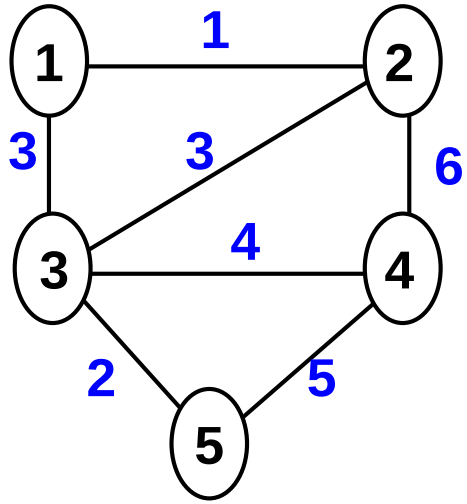
~~(V_1, V_3): 3~~

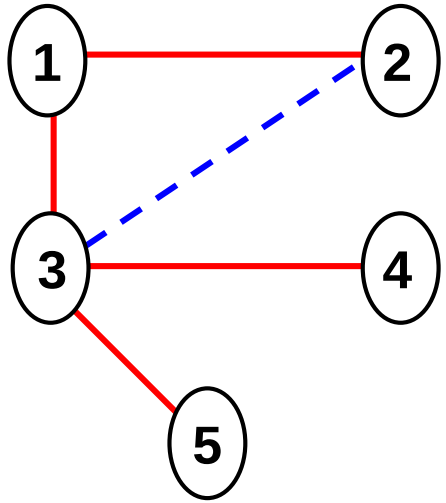
~~(V_2, V_3): 3~~

~~(V_3, V_4): 4~~

(V_4, V_5): 5

(V_2, V_4): 6





Kruskal's Algorithm

- Main idea

$T(n) = O(E \log E)$

```
Kruskal(G, w)
{
    A = ∅;
    for each vertex v ∈ V[G])
        MakeSet(v);
    sort the edges of E into increasing order by weight w
    for each edge (u, v) ∈ E, taken in increasing order by w
    {
        if FindSet(u) != FindSet(v) {
            A = A ∪ {(u, v)};  Union(u, v);
        }
    }
}
```

Prim算法

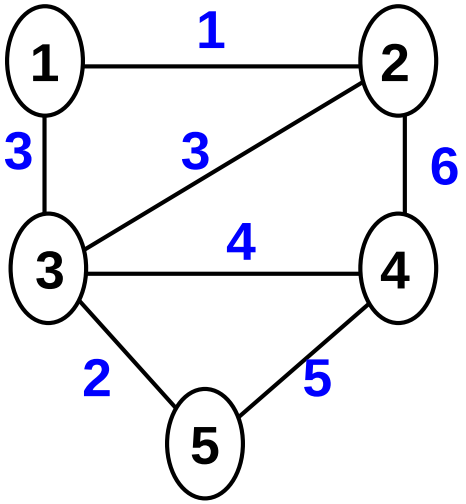
设$G=(V,E)$是一个加权连通图，其权函数为w,并设u为G的任意一个顶点.

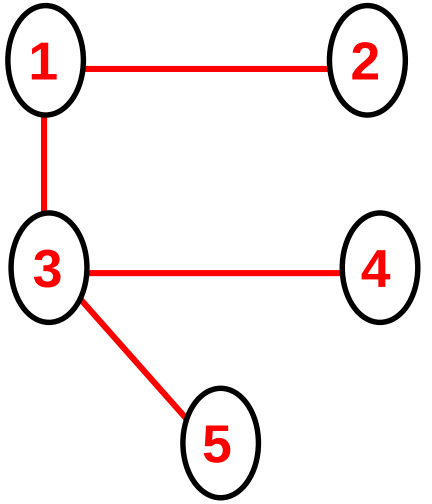
- 令$i=1, U_1=\{u\}, F_1=\Phi, T_1=(U_1,F_1)$.
- 对$i=1,2,\ldots,n-1$,执行：
  - 找出一个权最小的边$\alpha_i=\{x,y\}$,使得x在U_i中,而y不在U_i中.
  - 令$U_{i+1}=U_iU\{y\}, Fi_{+1}=F_iU\{\alpha_{i+1}\}$及$T_{i+1}=(Ui_{+1},Fi_{+1})$.
  - i值增加1
- 输出$T_{n_1}=(Un_{_1},Fn_{_1})$(这里$U_{n_1}=V$)

算法特点： 每一步，只需确定一条权最小的边，它连接一个已经到达顶点和一个未到达顶点，自动避免了圈的产生。

Prim's Algorithm

U	V−U
{1}	{2, 3, 4, 5}
{1, 2}	{3, 4, 5}
{1, 2, 3}	{4, 5}
{1, 2, 3, 5}	{4}
{1, 2, 3, 4, 5}	{}





Prim's Algorithm

- Main idea

$T(n) = O(E \log V)$

```
Prim(G, w)
{
    U = {1};
    while (1)
    {
        select w in V − U that is the nearest to U
        U = U ∪ {w};
        if ( U == V )  // solved
            break;
    }
}
```

集合U仅形成单棵树，添加入集合U的安全边总是连接树与一个不在树中的顶点的最小权边。

Outline

- 图的基本概念
- 图的基本算法
  - 图的遍历
    - **Breadth-first search**（广度优先搜索）
    - **depth-first search** （深度优先搜索）
  - **Minimum spanning trees (最小生成树)**
  - **Shortest Paths（最短路径）**
    - **Single-source shortest paths (单源点最短路径)**
    - **All-pairs shortest paths (所有结点间最短路径)**
  - **Max flow(最大流)**

最短路径问题

- 最短路径问题是图的一个比较典型的应用问题。
- 例如，某一地区的一个公路网，给定了该网内的n个城市以及这些城市之间的相通公路的距离，能否找到城市A到城市B之间一条距离最近的通路呢？如果将城市用点表示，城市间的公路用边表示，公路的长度作为边的权值，那么，这个问题就可归结为在网图中，求点A到点B的所有路径中，边的权值之和最短的那一条路径。这条路径就是两点之间的最短路径。

最短路径问题

- 在非网图中，最短路径是指两点之间经历的边数最少的路径。
- 对于网(带权图)来说，两个顶点之间的路径长度是路径中“弧的权值之和”。则当两个顶点之间存在多条路径时，其中必然存在一条“最短路径”，即路径中弧的权值和取最小值的那条路径。

Path in Graph

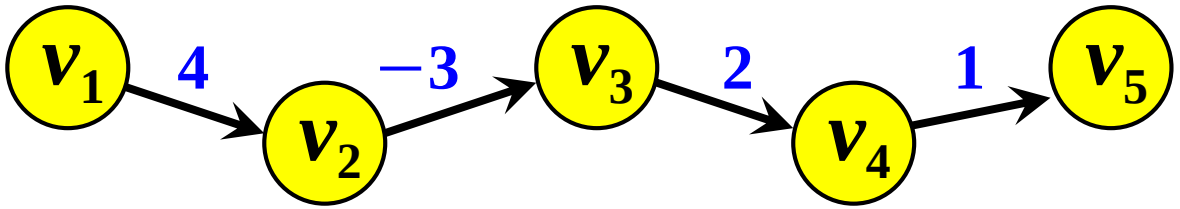
- Consider a digraph $G(V, E)$ with edge-weight function

$$w : E \rightarrow \mathbb{R}$$

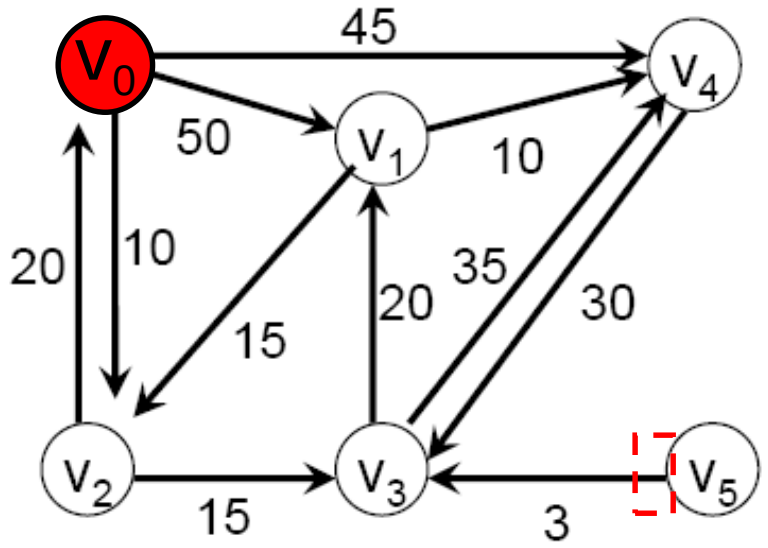
The weight of path $p = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$ is defined as

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

- Example



$w(p) = 4$



path	weight
(1) V_0V_2	10
(2) $V_0V_2V_3$	25
(3) $V_0V_2V_3V_1$	45
(4) V_0V_4	45

最短路径问题

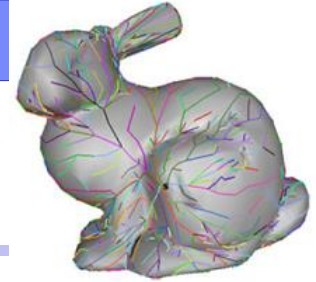
- 考虑到交通图的有向性，我们将讨论带权有向图，并称路径中的第一个顶点为“源点”，路径中的最后一个顶点为“终点”。

  以下讨论两种最常见的路径问题。

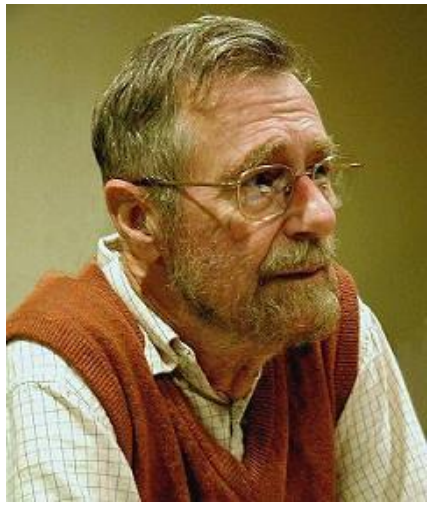
  1、求从某个源点到其余各点的最短路径；

  2、每一对顶点之间的最短路径。

Single-source Shortest Paths



- Single-source shortest path problem
  - Given a digraph $G(V, E)$, a source vertex s
  - Goal: find the shortest path from s to each vertex $v \in V$
- Dijkstra's Algorithm ---- 1959
  - by greedy method



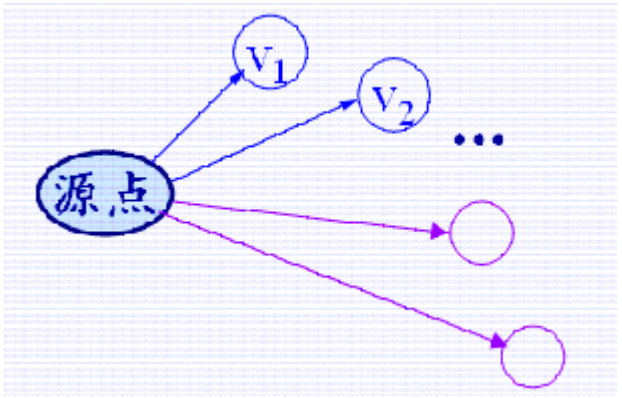
Dijkstra 1930~2002
荷兰计算机科学家
1972年获图灵奖

Dijkstra's Algorithm

算法的基本思想：

依最短路径长度递增的次序，求得各条路径。

假设图中所示为从源点到其余各点之间的最短路径，则在这些路径中，必然存在一条长度最短者。



Dijkstra's Algorithm

路径长度最短的最短路径的特点 (Observation)：

- 在这条路径上，必定只含一条（权值最小）弧，由此，只要在所有以源点出发的弧中查找权值最小者。
- 长度最短的路径可能有两种情况：

  它可能是从源点直接到该点的路径；

  也可能是，从源点经过顶点v_1，再到达该顶点。
- 其余最短路径的特点：

  它或者是直接从源点到该点(只含一条弧)； 或者是，从源点经过已求得最短路径的顶点，再到达该顶点。

Dijkstra's Algorithm

假设Dist[k]表示当前所求得的从源点到顶点k的最短路径

则一般情况下，

Dist[k]＝〈源点到顶点k的弧上的权值〉

或者＝〈源点到其它顶点的路径长度〉＋〈其它顶点到顶点k的弧上的权值〉

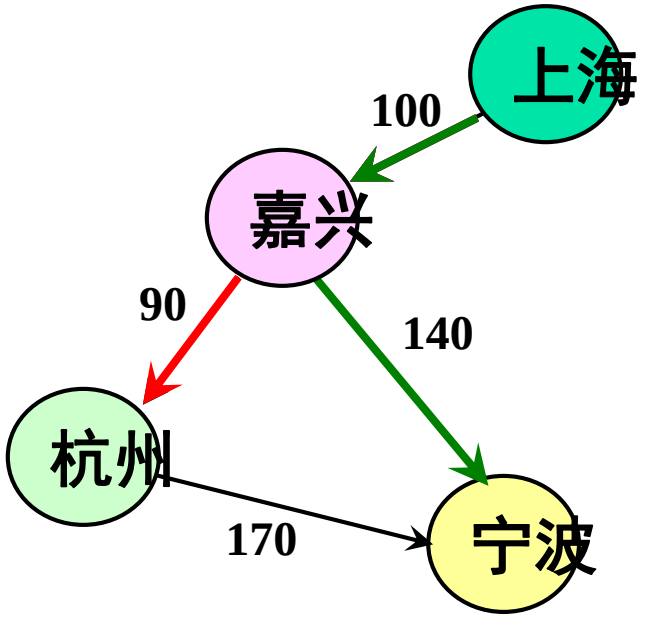
Dijkstra's Algorithm

- Data structure
  - **float w[1…n][1…n]**
    - $G = (V, E)$, $|V| = n$

$$w[i][j] = \begin{cases} \text{edge weight} & \langle i, j\rangle \in E \\ 0 & i = j \\ \infty & \langle i, j\rangle \notin E \end{cases}$$

  - **bool F[1…n]**
    - **F[i] = 0**: default value
    - **F[i] = 1**: 源点到结点 v_i 的最短路径已确定
  - **Dist[1…n]**
    - 源点到结点 v_i 的最短路径

Example



w[4][4]	上海	嘉兴	杭州	宁波
上海	0	100	∞	∞
嘉兴	∞	0	90	140
杭州	∞	∞	0	170
宁波	∞	∞	∞	0

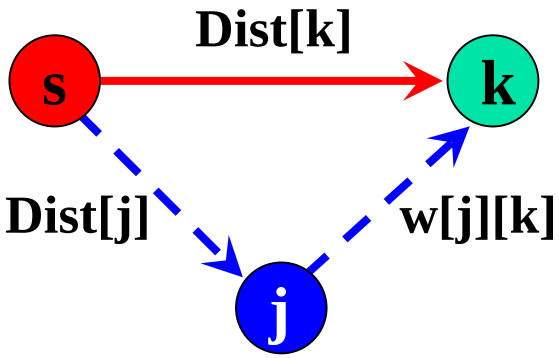
F = [上海 嘉兴 杭州 宁波]

Dist = [0 100 190 240]

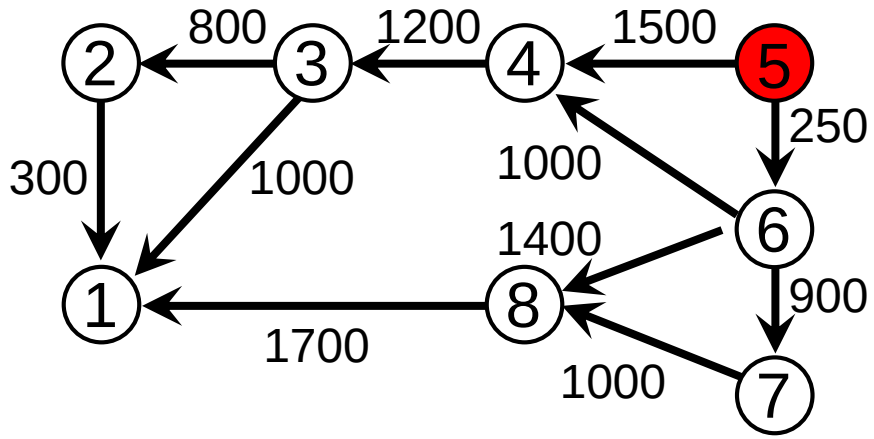
Dijkstra's Algorithm

1. Dist[1…n] = w[s][1…n], 即 w 的第 s 行 // s 为源点
2. F[1…n] = 0, F[s] = 1 // F[i] = 1: 已确定从 s到结点 i 的最短路径
3. 从尚未确定最短路径的所有结点 (F[i] = 0) 中, 找出 Dist[j] 最小的 j, 令 F[j] = 1
4. 对 F[k] = 0 的所有结点 k, 更新 Dist[k]

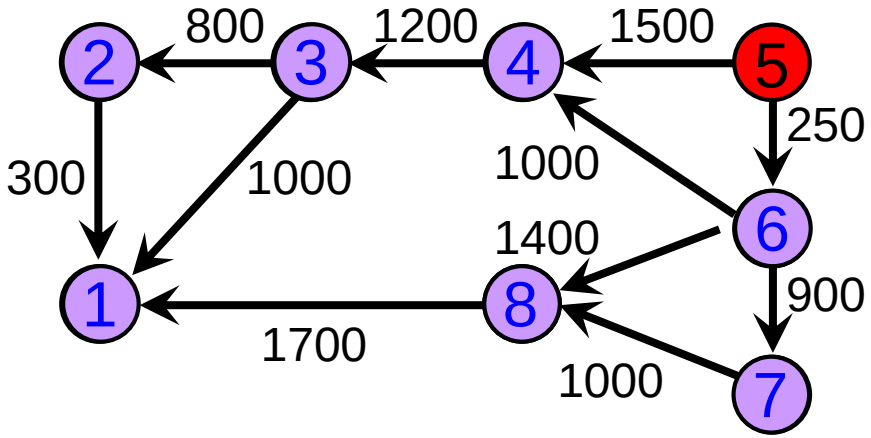
   Dist[k] = min{Dist[k], Dist[j] + w[j][k]}



5. 重复步骤3-4, 直到所有的 F[i] = 1



w[i][j]	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1700
8	1700							0



	1	2	3	4	5	6	7	8
F	1	1	1	1	1	1	1	1

状态	选择结点 j	Dist 1	2	3	4	5	6	7	8
初始									
1									
2									
3									
4									
5									
6									
7									

Dijkstra's Algorithm

```
void Dijkstra(float w[][], float Dist[], int n, int s) // w[1..n][1..n], Dist[1..n]
{
    for (int i = 1; i <= n; i++)
        F[i] = 0;  Dist[i] = w[s][i];
    F[s] = 1; Dist[s] = 0;

    for (i = 1; i < n; i++)  // loop n−1 times
    {
        int j = 0; float mindst = INF;
        for (int k = 2; k <= n; k++) // step 3
            if (F[k] == 0 && mindst > Dist[k])
                j = k; mindst = Dist[k];
        F[j] = 1;
        for (k = 1; k <= n; k++)      // step 4
            if (F[k] == 0 && Dist[j] + w[j][k] < Dist[k])
                Dist[k] = Dist[j] + w[j][k];
    }
}
```

$T(n) = O(n^2)$

Shortest Paths : Implementation

```
template <class Weight, int graph_size>
class Digraph {
public:                    //  Add a constructor and methods for Digraph input and output.
   void set_distances(Vertex source, Weight distance[ ]) const;
protected:
   int count;
   Weight adjacency[graph_size] [graph_size];
};
```

```
template <class Weight, int graph_size>
void Digraph<Weight, graph_size> :: set_distances(Vertex source,
                                    Weight distance[ ]) const
/* Post: The array distance gives the minimal path weight from vertex source to each
         vertex of the Digraph. */
{ Vertex v, w; bool found[graph_size]; //  Vertices found in S
  for (v = 0; v < count; v++) {
    found[v] = false;
    distance[v] = adjacency[source][v];
  }
  found[source] = true;        //  Initialize with vertex source alone in the set S.
  distance[source] = 0;
  for (int i = 0; i < count; i++) {  //  Add one vertex v to S on each pass.
    Weight min = infinity;
    for (w = 0; w < count; w++) if (!found[w])
      if (distance[w] < min) {
        v = w;
        min = distance[w];
      }
    found[v] = true;
    for (w = 0; w < count; w++) if (!found[w])
      if (min + adjacency[v][w] < distance[w])
        distance[w] = min + adjacency[v][w];
  }
}
```

初始化 distance

初始化found

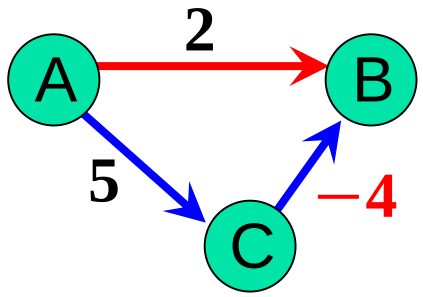
找出下一个加入S的结点v

将v加入S后， 修改数组 found 和distance

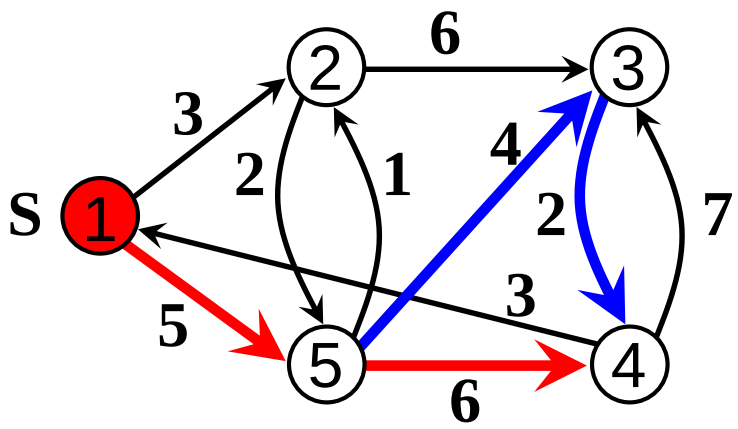
时间复杂度：$O(n^2)$

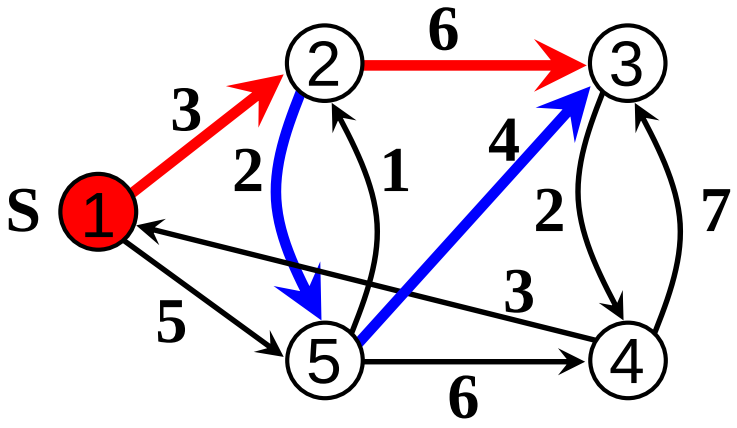
Notice

- Dijkstra's algorithm cann't be used in graph with negative weight!



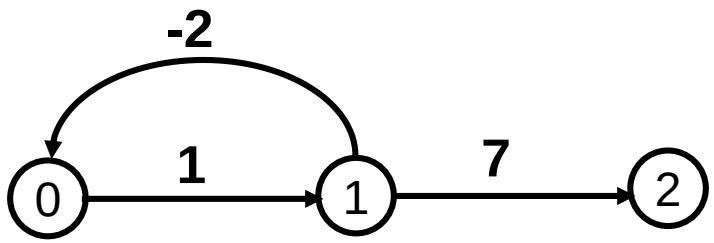
- Shortest paths are not necessarily unique





Bellman-Ford算法

- 为了能够求解边上带有负值的单源最短路径问题，Bellman(贝尔曼)和Ford(福特)提出了从源点逐次绕过其他顶点，以缩短到达终点的最短路径长度的方法。
- **Bellman-Ford算法的限制条件：**
  - 要求图中不能包含权值总和为负值回路(负权值回路)，如下图所示。Why?



(c)

如果存在从源点可达的负权值回路，则最短路径不存在，因为可以重复走这个回路，使得路径无穷小。

Bellman-Ford算法思想

- Bellman-Ford算法构造一个最短路径长度数组序列dist ^{1} [u], dist ^{2} [u], …, dist $^{n-1}$ [u]。其中：
- dist ^{1} [u]为从源点v到终点u的只经过一条边的最短路径长度，并有dist ^{1} [u] =Edge[v][u]；
- dist ^{2} [u]为从源点v最多经过两条边到达终点u的最短路径长度；
- dist ^{3} [u]为从源点v出发最多经过不构成负权值回路的三条边到达终点u的最短路径长度；
- ……
- dist $^{n-1}$ [u]为从源点v出发最多经过不构成负权值回路的n-1条边到达终点u的最短路径长度；
- 算法的最终目的是计算出dist $^{n-1}$ [u]，为源点v到顶点u的最短路径长度。

Bellman-Ford算法思想

- 采用递推方式计算 $dist^k[u]$。
  - 设已经求出 $dist^{k-1}[u]$, $u = 0, 1, \ldots, n-1$，此即从源点v最多经过不构成负权值回路的$k-1$条边到达终点u的最短路径的长度。
  - 从图的邻接矩阵可以找到各个顶点j到达顶点u的距离Edge[j][u]，计算 $\min\{ dist^{k-1}[j] + Edge[j][u] \}$，可得从源点v绕过各个顶点，最多经过不构成负权值回路的k条边到达终点u的最短路径的长度。
  - 比较$dist^{k-1}[u]$和$\min\{ dist^{k-1}[j] + Edge[j][u] \}$，取较小者作为$dist^k[u]$的值。

Bellman-Ford算法思想

递推公式(求顶点u到源点v的最短路径)：

$dist^1[u] = Edge[v][u]$

$dist^k[u] = \min\{ dist^{k-1}[u], \min\{ dist^{k-1}[j] + Edge[j][u] \} \}$, j=0,1,…,n-1,j≠u

算法实现

注意：

1. 本算法使用同一个数组***dist*** **[u]**来存放一系列的$dist^k$**[u]**。其中***k = 0, 1, …, n-1***。算法结束时中存放的是$dist^{n-1}$**[u]**。
2. ***path***数组含义同**Dijkstra算法**中的***path***数组。

```
#define MAX_VER_NUM 10     //顶点个数最大值
#define MAX 1000000
int Edge[MAX_VER_NUM][MAX_VER_NUM];  //图的邻接矩阵
int vexnum;      //顶点个数

void BellmanFord(int v) //假定图的邻接矩阵和顶点个数已经读进来了
{
        int i, k, u;
        for(i=0; i<vexnum; i++)
        {
                dist[i]=Edge[v][i];        //对dist[ ]初始化
                if( i!=v && dis[i]<MAX ) path[i] = v;      //对path[ ]初始化
                else path[i] = -1;
        }
```

```
        for(k=2; k<vexnum; k++) //从dist[1][u]递推出dist[2][u], …,dist[n-1][u]
        {
                for(u=0; u< vexnum; u++)//修改每个顶点的dist[u]和path[u]
                {
                        if( u != v )
                        {
                                for(i=0; i<vexnum; i++)//考虑其他每个顶点
                                {
                                        if( Edge[i][u]<MAX &&
                                            dist[u]>dist[i]+Edge[i][u] )
                                        {
                                                dist[u]=dist[i]+Edge[i][u];
                                                path[u]=i;
                                        }
                                }
                        }
                }
        }
}
```

如果dist[ ]各元素的初值为MAX，则循环应该n-1次，即k的初值应该改成1。

Dijkstra算法与Bellman算法的区别

- Dijkstra算法和Bellman算法思想有很大的区别：
  - Dijkstra算法在求解过程中，源点到集合S内各顶点的最短路径一旦求出，则之后不变了，修改的仅仅是源点到T集合中各顶点的最短路径长度。
  - Bellman算法在求解过程中，每次循环都要修改所有顶点的dist[ ]，也就是说源点到各顶点最短路径长度一直要到Bellman算法结束才确定下来。

判断负值回路的方法

思路：在求出$dist^{n-1}[\ ]$之后，再对每条边<u,v>判断一下：加入这条边是否会使得顶点v的最短路径值再缩短，即判断：

dist[u]+w(u,v)<dist[v]

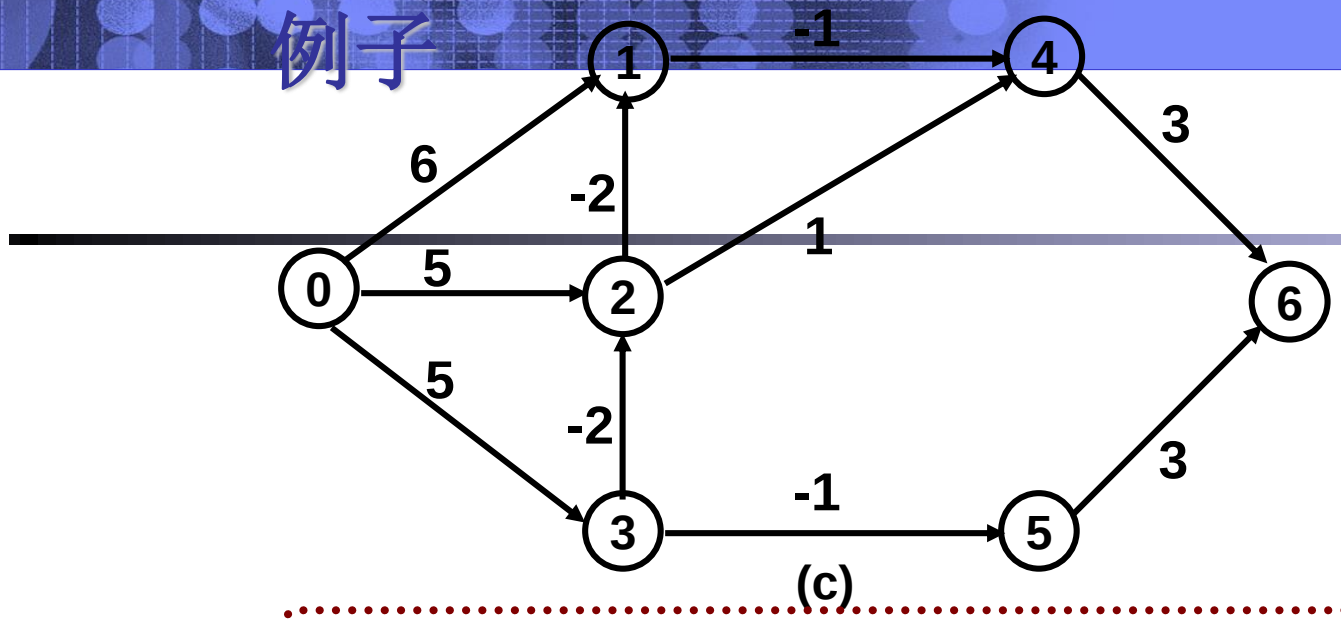
是否成立，如果成立，则说明存在从源点可达的负权值回路。代码如下：

```
for (i=0;i<n;i++)
{
        for (j=0;j<n;j++)
        {
                if (Edge[i][j]<MAX && dist[j]>dist[i]+Edge[i][j])
                        return 0;//存在从源点可达的负权值回路
        }
}
return 1;
```

算法复杂度分析

- 假设图的顶点个数为n，边的个数为e。算法中有一个三重嵌套的for循环，如果：
  - 使用邻接矩阵存储图，最内层if语句的总执行次数为$O(n^3)$，因此算法的复杂度为$O(n^3)$；
  - 使用邻接表存储图，内层的两个for循环改成while循环，可以使算法的复杂度降为O(n*e)。

例子



$$\begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{bmatrix} 0 & 6 & 5 & 5 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & -1 & \infty & \infty \\ \infty & -2 & 0 & \infty & 1 & \infty & \infty \\ \infty & \infty & -2 & 0 & \infty & -1 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & \infty & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{bmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} \end{array}$$

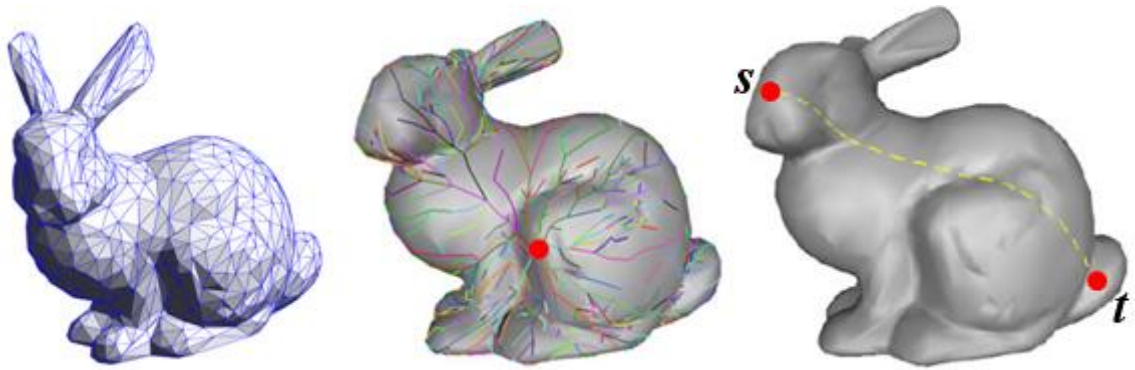
$dist^2[1] = \min\{ dist^1[1], \min\{ dist^1[j] + Edge[j][1] \} \}$

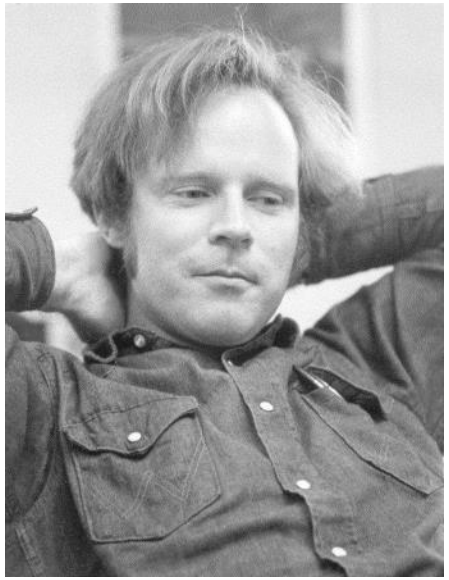
$= \min\{6, \min\{dist^1[0]+Edge[0][1], dist^1[2]+Edge[2][1], \ldots \} \}$

k	$dist^k[0]$	$dist^k[1]$	$dist^k[2]$	$dist^k[3]$	$dist^k[4]$	$dist^k[5]$	$dist^k[6]$
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

All-pairs Shortest Paths

- All-pairs shortest path problem
  - Given a digraph $G(V, E)$
  - Goal: find the shortest path from any vertex s to t





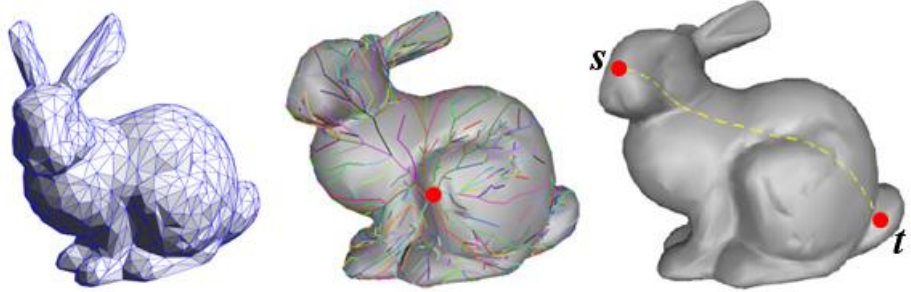
Robert W. Floyd

1936~2001

美国计算机科学家

1978年获图灵奖

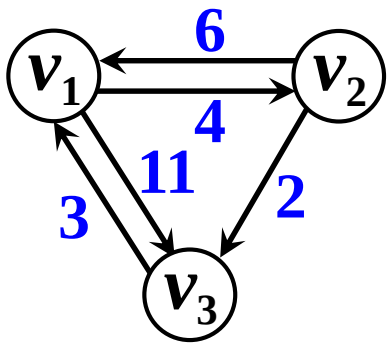
The Problem



- Shortest path problem
  - Single-source
    - Use Dijkstra's algorithm by greedy method
    - $T(n) = O(n^2)$
  - All-pairs
    - Use Dijkstra's algorithm n times
      - $T(n) = O(n^3)$
    - Use Floyd's Algorithm by dynamic programming
      - $T(n) = O(n^3)$

Floyd's Algorithm

- $D^k[i][j]$
  - V_i 到 V_j 的最短路径, 且途中经过结点的编号都 ≤ k (k≥0)



$D^1[2, 1] = 6$

$2 \rightarrow 1$

$D^2[2, 1] = 6$

$2 \rightarrow 1$

$2 \rightarrow 1 \rightarrow 2 \rightarrow 1$

$D^3[2, 1] = 5$

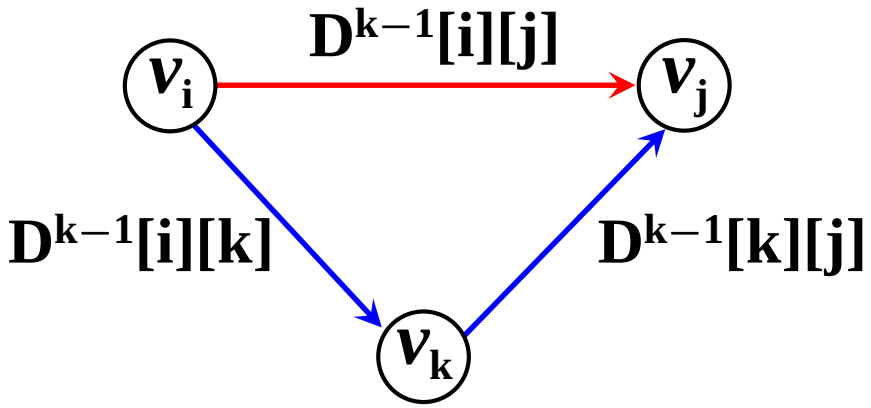
$2 \rightarrow 1$

$2 \rightarrow 1 \rightarrow 2 \rightarrow 1$

$2 \rightarrow 3 \rightarrow 1$

Floyd's Algorithm

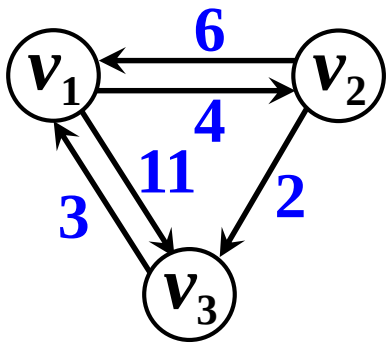
$$D^k[i][j] = \min \begin{cases} D^{k-1}[i][j] \\ D^{k-1}[i][k] + D^{k-1}[k][j] \end{cases}$$



Floyd's Algorithm

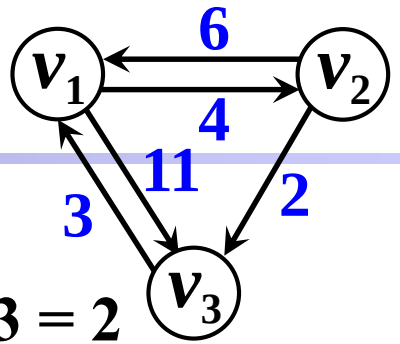
- **k = 0, $D^k[i][j]$** is the weight from V_i to V_j

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix} \end{array}$$



- Steps
  1. **D^0 = weight matrix**
  2. **Compute $D^1, D^2, \ldots, D^n$**
  3. **D^n is the solution**

$$D^k[i][j] = \min \begin{cases} D^{k-1}[i][j] \\ D^{k-1}[i][k] + D^{k-1}[k][j] \end{cases}$$



- Compute D^1, D^2, D^3

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix} \end{array}$$

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \end{array}$$

$2 \to 3 = 2$

$2 \to 1 \to 3 = 17$

$3 \to 2 = \infty$

$3 \to 1 \to 2 = 7$

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \end{array}$$

$1 \to 3 = 11$

$1 \to 2 \to 3 = 6$

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \end{array}$$

$1 \to 2 = 4$

$1 \to 3 \to 2 = \infty$

$2 \to 1 = 6$

$2 \to 3 \to 1 = 5$

$3 \to 1 = 3$

$3 \to 2 \to 1 = \infty$

Floyd's Algorithm

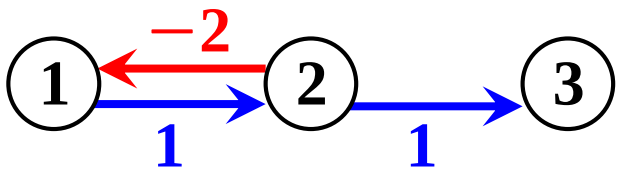
```
void Floyd(float w[][], float D[][], int n)
// w[1...n][1...n], D[1...n][1...n]
{
    D = w;  // D[i][j] = w[i][j]

    for (int k = 1; k <= n; k++)    // 计算D^k
    {
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
    }
}
```

$T(n) = O(n^3)$

Notice

- Floyd's algorithm cann't be used in graph having a cycle with negative weight!



$$D^2[1][3] \neq \min\{D^1[1][3], D^1[1][2] + D^1[2][3]\} = 2$$

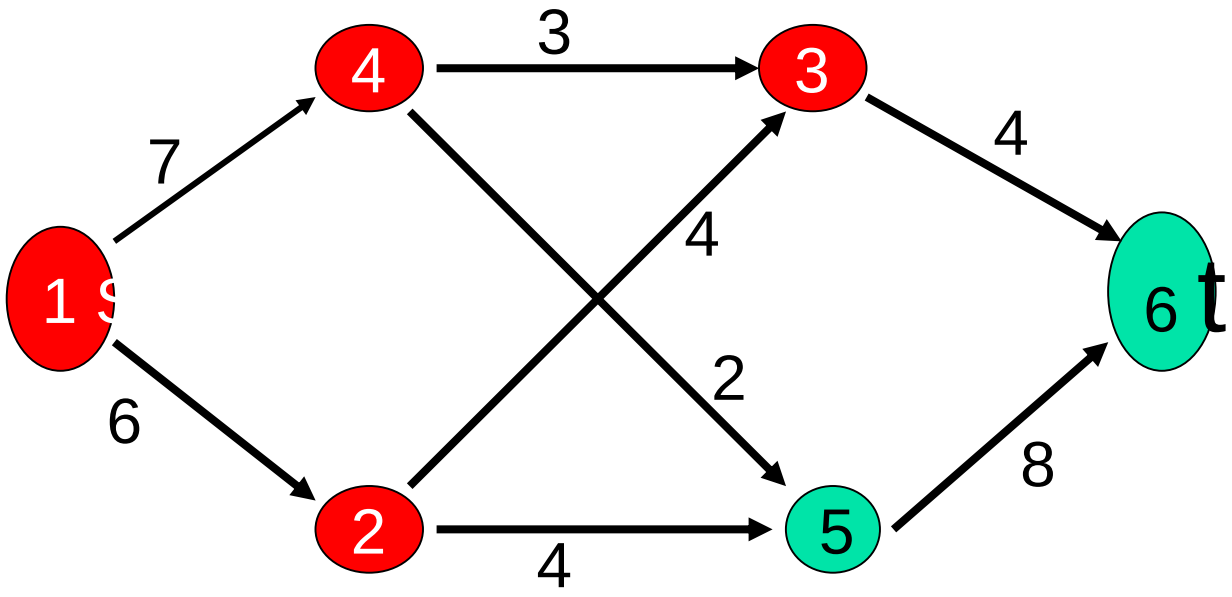
Outline

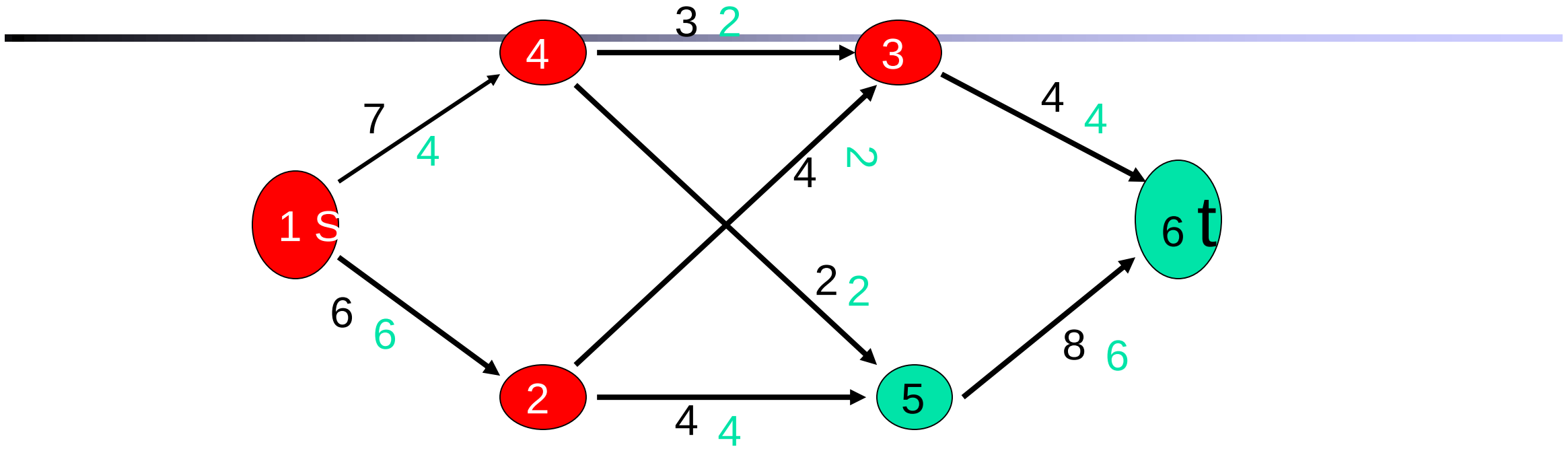
- 图的基本概念
- 图的基本算法
  - 图的遍历
    - **Breadth-first search**（广度优先搜索）
    - **depth-first search** （深度优先搜索）
  - **Minimum spanning trees (最小生成树)**
  - **Shortest Paths**（最短路径）
    - **Single-source shortest paths (单源点最短路径)**
    - **All-pairs shortest paths (所有结点间最短路径)**
  - **Max flow(最大流)**

实例

有一自来水管道输送系统，起点是S，目标是T，途中经过的管道都有一个最大的容量。

- 问题：问从S到T的最大水流量是多少？





最大水流量是10

网络流的定义

有向图 G = (V, E)中：

- 有唯一的一个源点S（入度为0：出发点）
- 有唯一的一个汇点 T（出度为0：结束点）
- 图中每条弧 (u, v) 都有一非负容量 c (u, v)

满足上述条件的图G称为网络流图。

记为： G = (V, E ,C)

可行流

◆每条弧 （ u, v ）上 给定一个实数f(u,v),满足：有 0 <= f （ u, v ） <= c(u, v),则f(u,v)称为弧(u, v)上的流量。

◆如果有一组流量满足条件：

源点s ： 流出量 = 整个网络的流量

汇点t ： 流入量 =整个网络的流量

中间点：总流入量 = 总流出量

那么整个网络中的流量成为一个可行流。

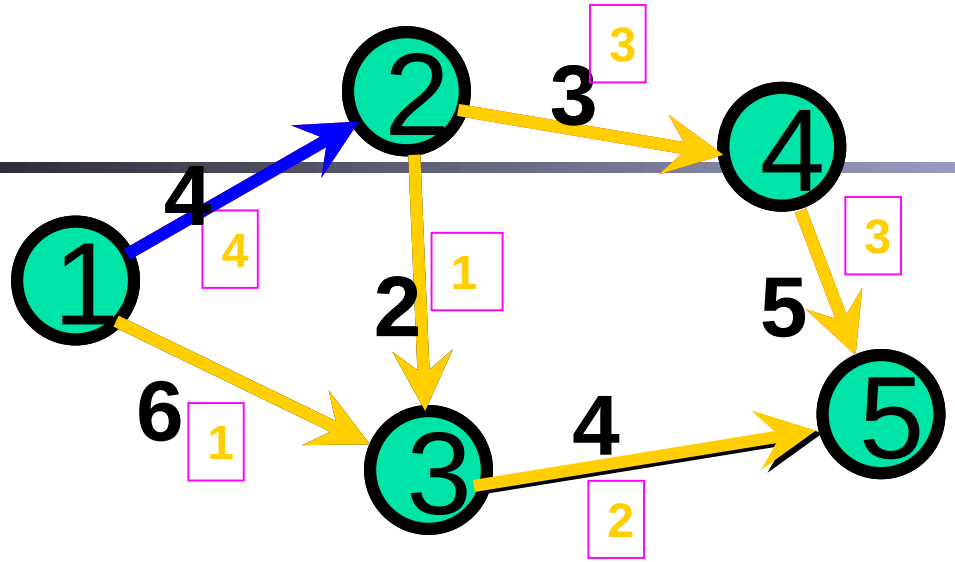


图1

一个可行流：5

一个可行流：7

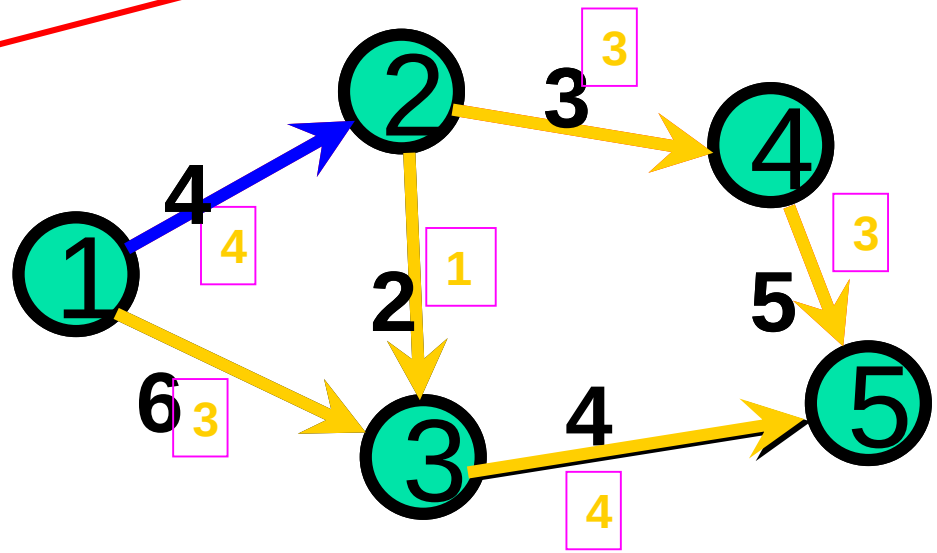


图2

最大流

- 在所有的可行流中， 流量最大的一个流的流量

  如： 图2中可行流7也是最大流。
- 最大流可能不只一个。

最大流算法

◆Ford-Fulkerson （福特-福克森）算法

步骤：

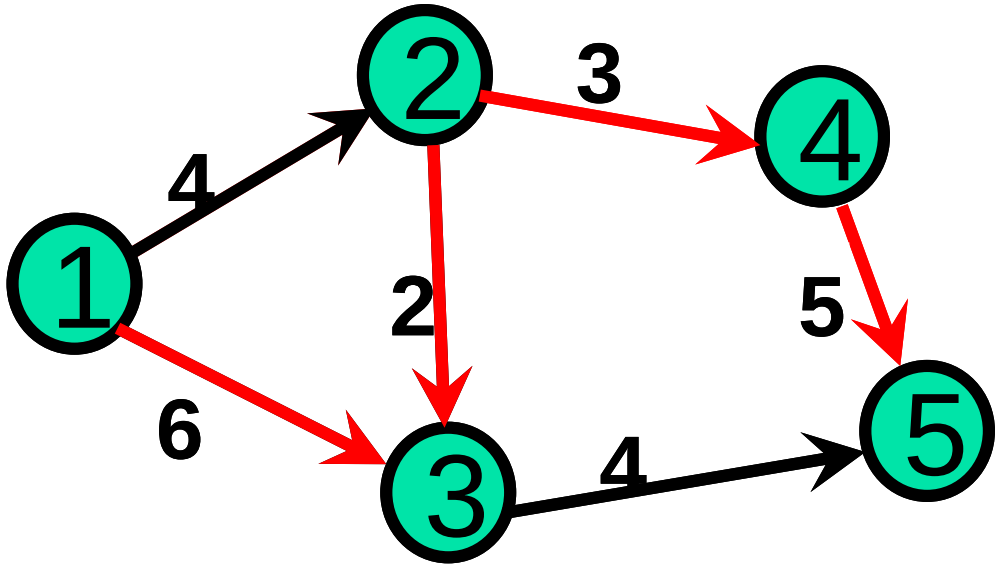
（1）如果存在增广路径,就找出一条增广路径

（2）然后沿该条增广路径进行更新流量

（增加流量）

增广路径

- 从 s 到 t 的一条简单路径，若边 (u, v) 的方向与该路径的方向一致，称 (u, v) 为正向边，方向不一致时称为逆向边。



简单路：1→3 → 2→4→5中。

（1，3）（2，4）（4，5）是正向边。（3，2）是逆向边。

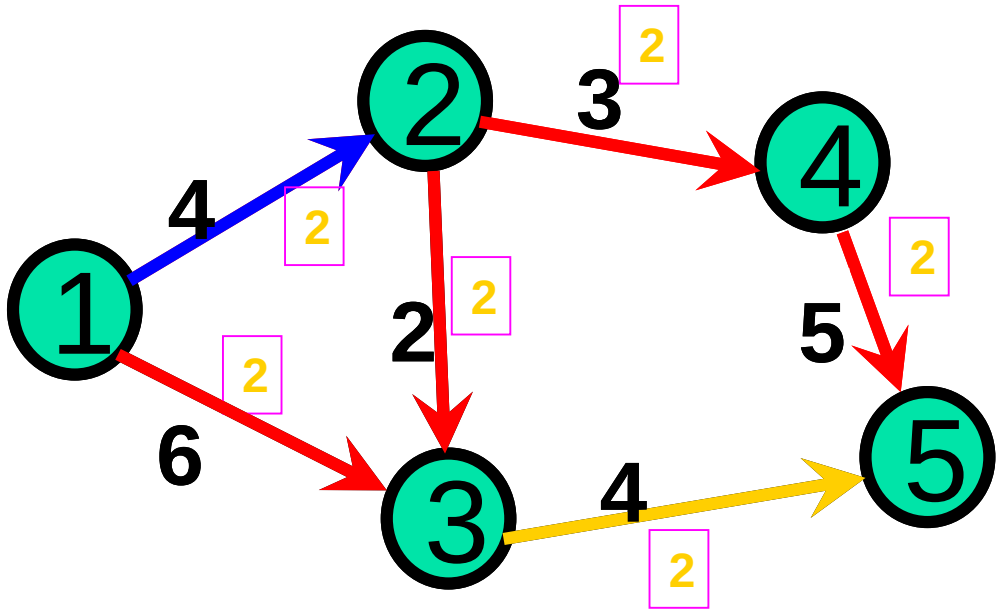
增广路径

若路径上所有的边满足：

①所有正向边有：f（u，v）＜c（u，v）

②所有逆向边有：f（u，v）＞0

则称该路径为一条增广路径(可增加流量)



两条增广路径：

1→3→5

1→3 → 2→4→5

增加流量=？

沿增广路径增广

1） 先设d为为正无穷(可增加流，余流量)

若(u, v) 是正向边

$d = \min (d, c (u, v) – f (u, v))$

若(u, v) 是逆向边

$d = \min (d, f (u, v))$

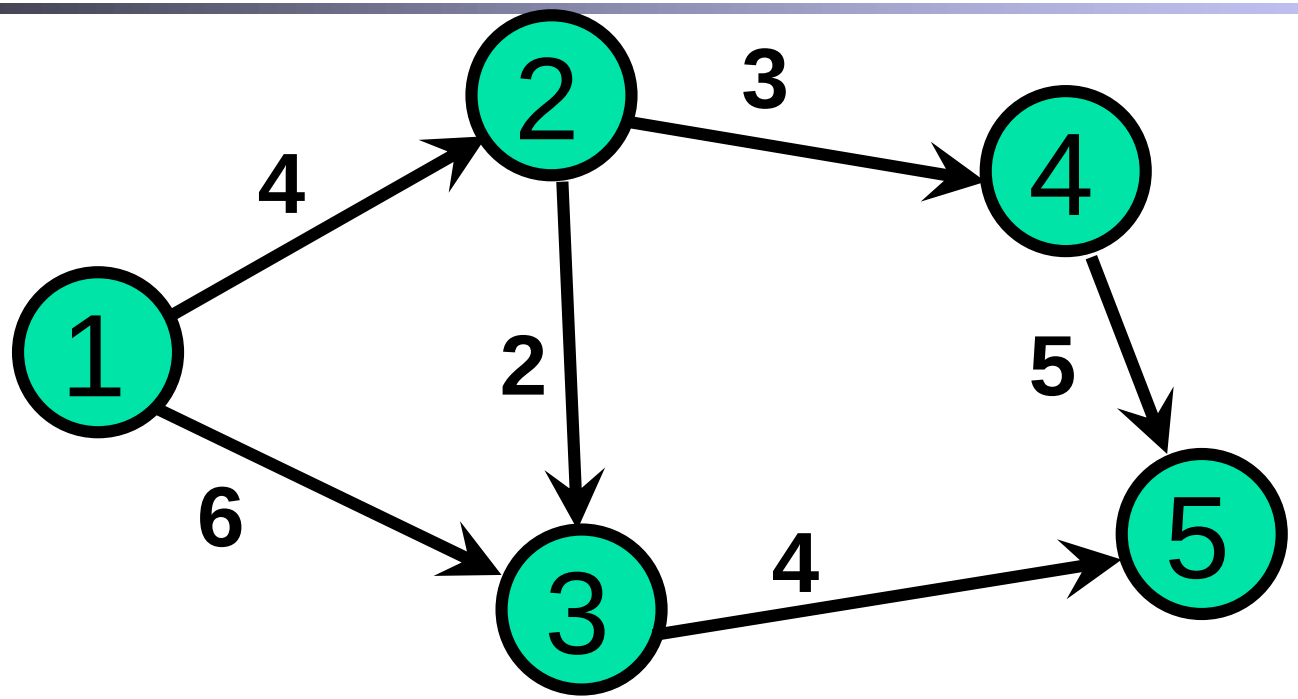
2 ） 对于该增广路径上的边

若(u, v) 是正向边，$f (u, v) = f (u, v) + d$;

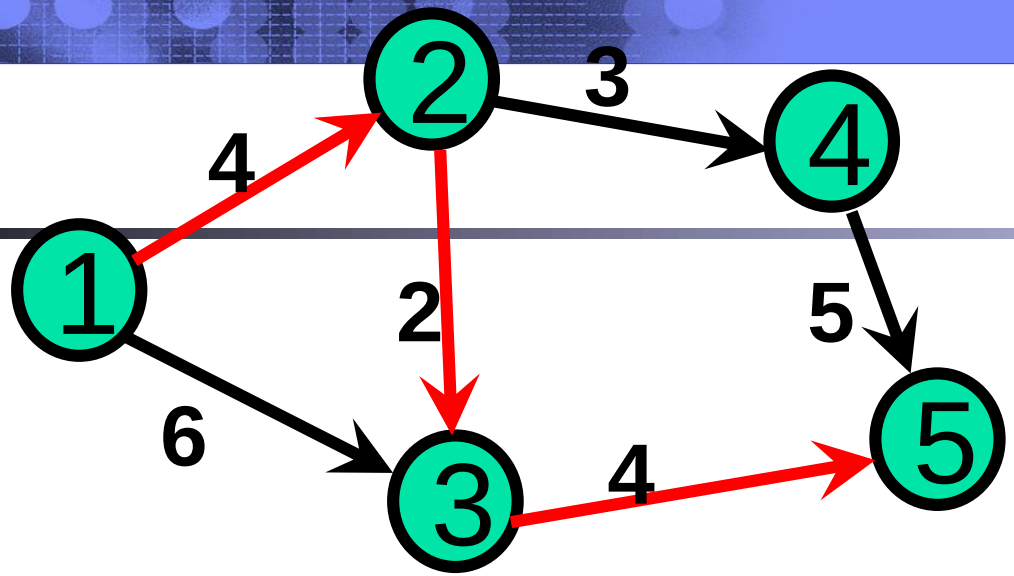
若(u, v) 是逆向边，$f (u, v) = f (u, v) – d$;

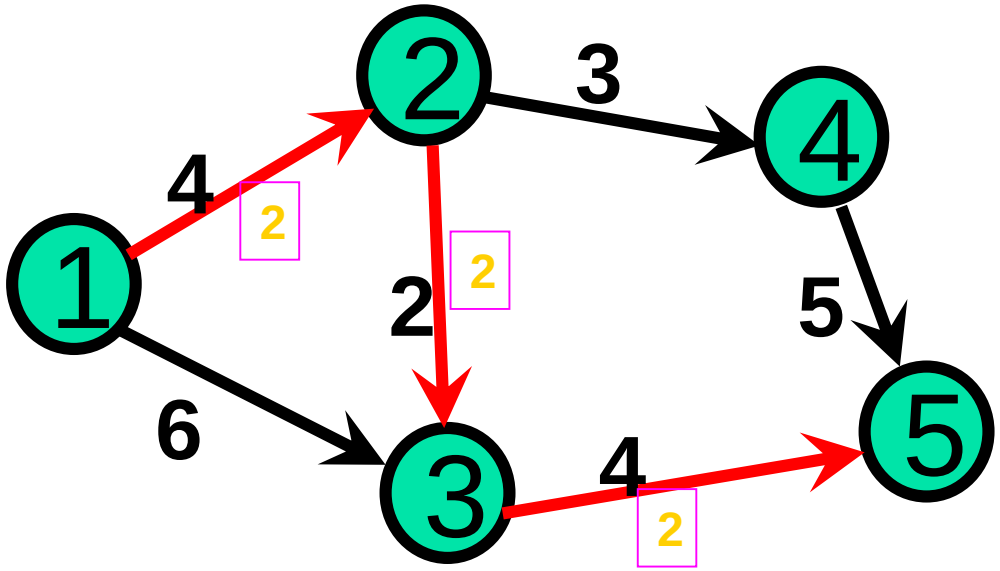
增广后，总流量增加了d

样例



开始流量为:sum=0





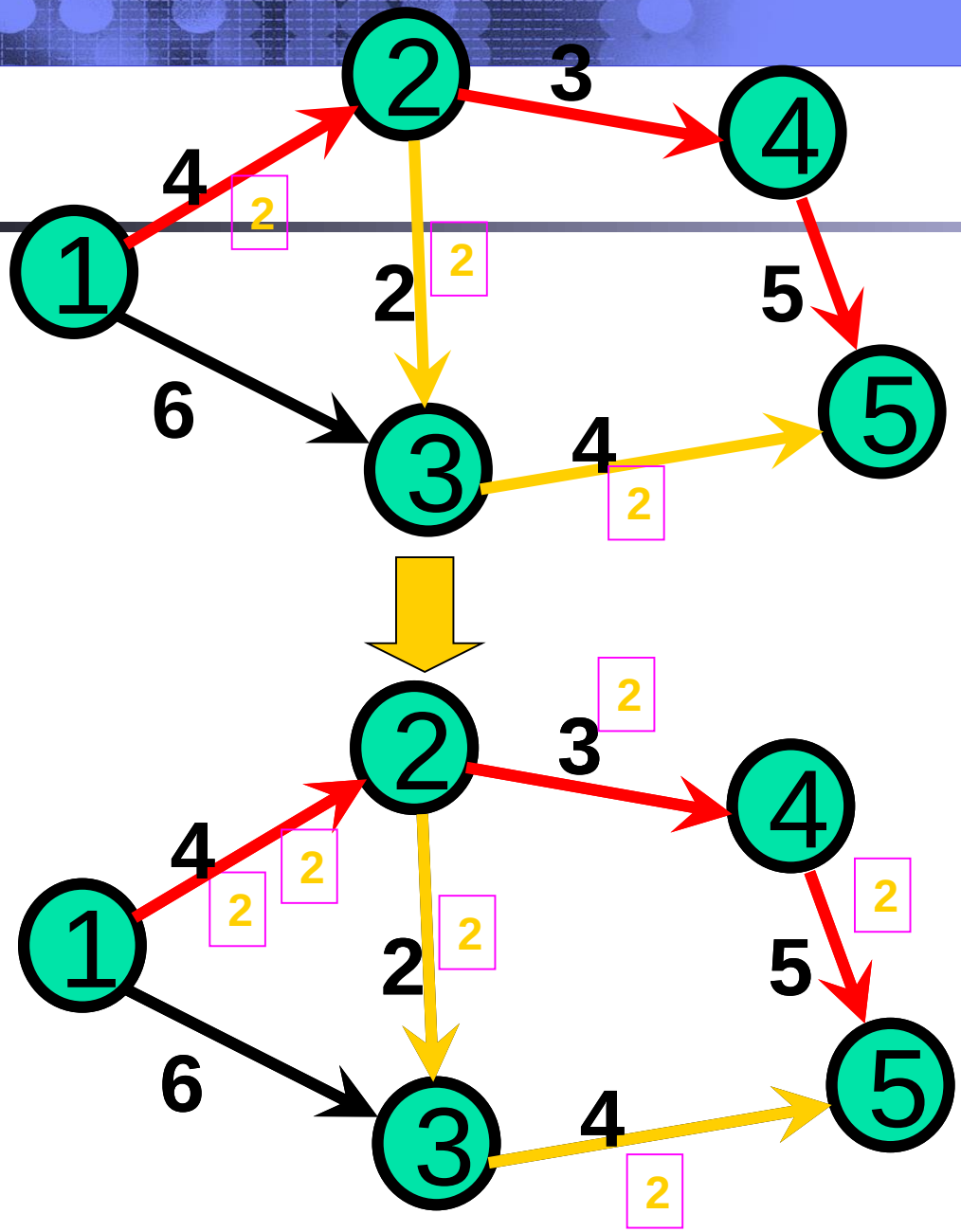
1、一条增广路径:

1→2→3→5

d=min{4,2,4} =2

增加流量: 2

Sum=2



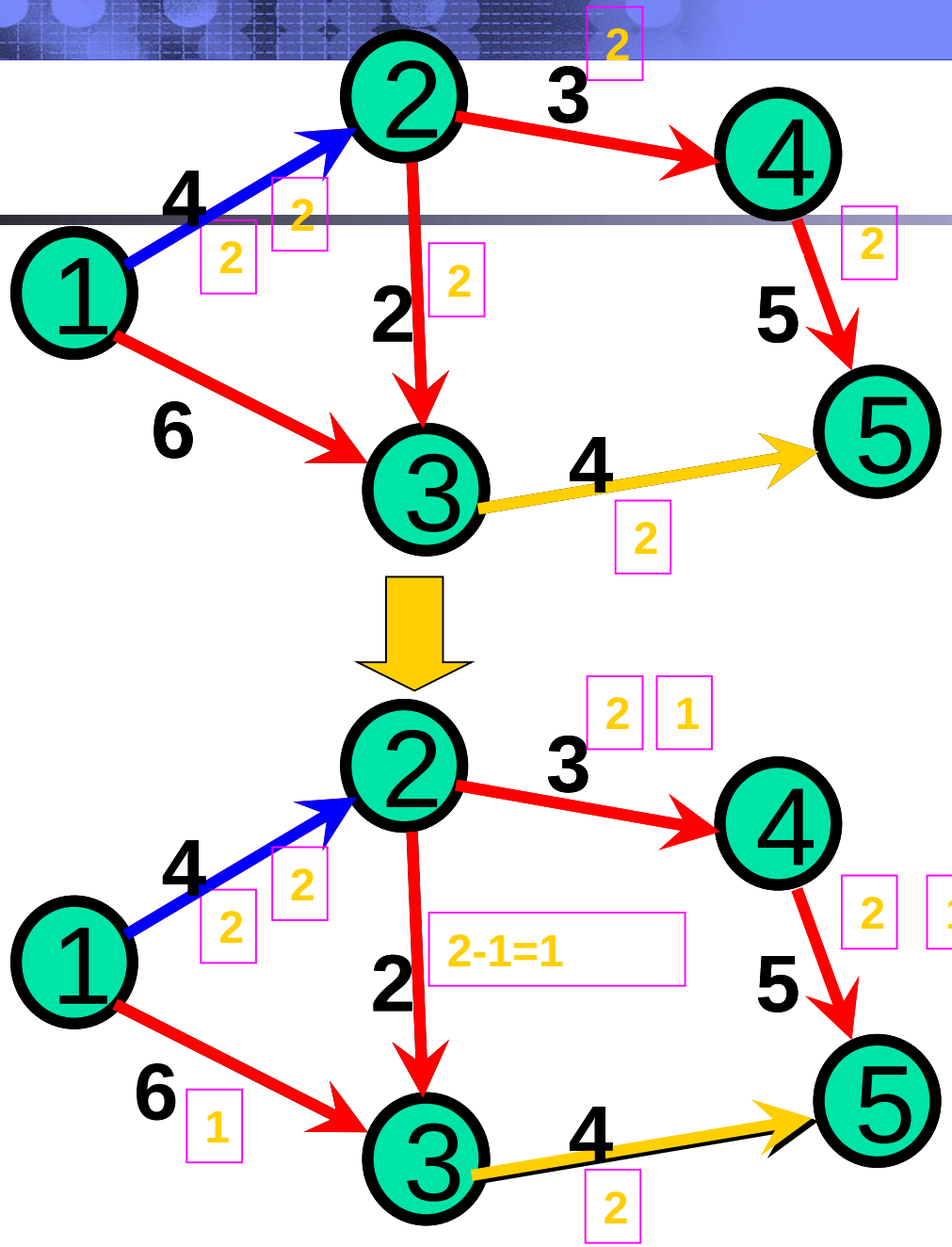
2、一条增广路径:

1→2→4→5

d=min{4-2,3,5} =2

增加流量: 2

Sum=2+2=4



3、一条增广路径:

1→ 3 → 2 → 4 →5

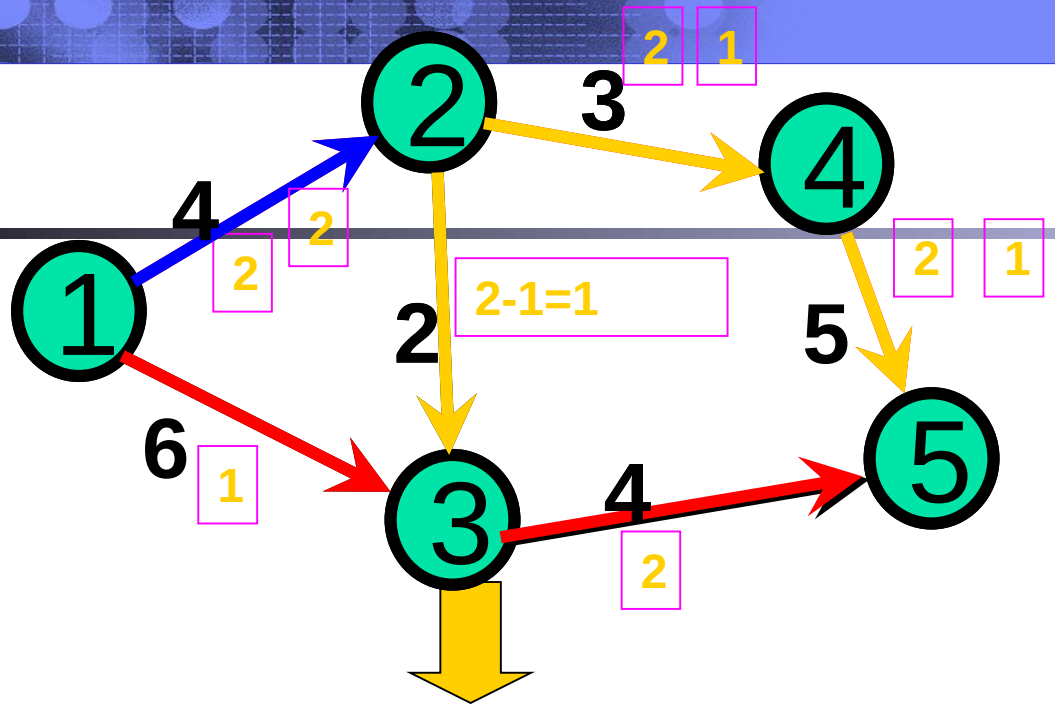
d=min{6,2,3-2,5-2} =1

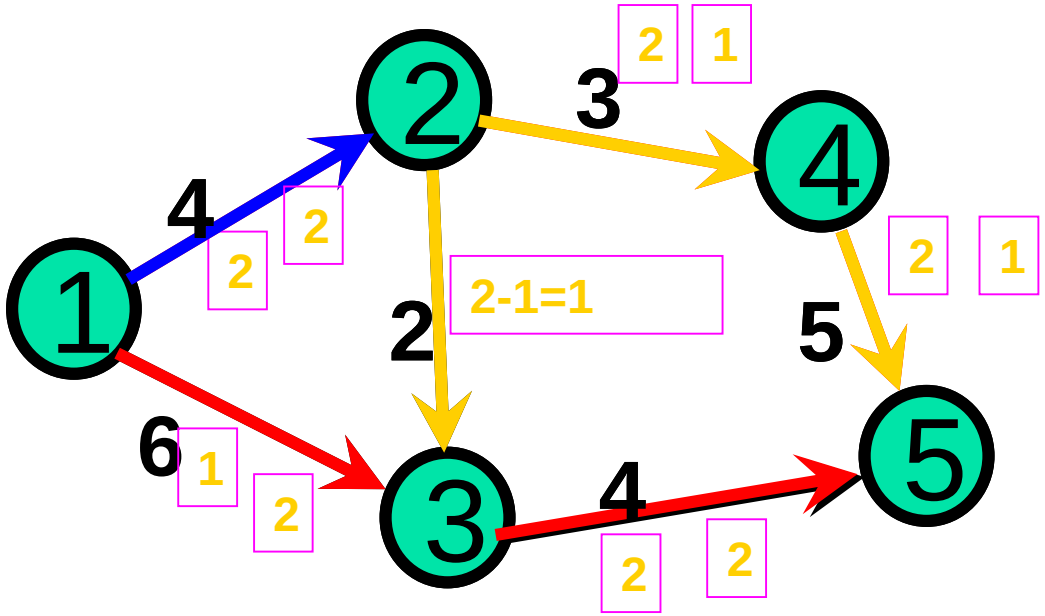
增加流量: 1

Sum=4+1=5

2->3减少 1， 加到 2 -> 4

2 ->3减的 1 由 1 -> 3 补充 1





4、一条增广路径:

1→ 3 →5

d=min{6-1,4-2} =2

增加流量: 2

Sum=5+2=7

定理：

可行流 f 为最大流，当且仅当不存在关于f 的增广路径

证：若 f 是最大流，但图中存在关于 f 的增广路径，则可以沿该增广路径增广，得到的是一个更大的流，与f 是最大流矛盾。所以，最大流不存在增广路径。

◆Ford-Fulkerson算法

步骤：

（1）如果存在增广路径，就找出一条增广路径

DFS，BFS

（2）然后沿该条增广路径进行更新流量

（增加流量）

While 有增广路径 **do** 更新该路径的流量

迭代算法

算法的实现

c [u, v]：容量

f [u, v]：流量

B[i]：保存找到的增广路径，记录路径上结点i的前驱结点。

Sum：最大流量。

假定：1是源点S；n是汇点T。

1）：DFS找增广路径

```
function findflow(k:integer):boolean;  {找结点k的后继结点i }
    var i:integer;
    begin
      if k=n then exit(true);      {找到了一条增广路径}
      for i:=1 to n do
        if(b[i]=-1) and((c[k,i]-f[k,i]>0)or(f[i,k]>0)) then
          begin
            b[i]:=k;
            if findflow(i) then exit(true);
          end;
      exit(false);
    end;
```

2） procedure addflow;//沿增广路径增广（增加流量）

```
d:=maxint; {增量}
 i:=n;  {沿增广路径的终点向起点反向求d}
 while b[i]<>0 do
    begin
       if c[b[i],i]>0 then d:=min(d,c[b[i],i]-f[b[i],i]); {正向边}
       if c[i,b[i]]>0 then d:=min(d,f[i,b[i]]);          {逆向边}
       i:=b[i];
    end;
 i:=n;
 while b[i]<>0 do {逆向更新每条边的流量}
    begin
       if c[b[i],i]>0 then inc(f[b[i],i],d); {正向边}
       if c[i,b[i]]>0 then dec(f[i,b[i]],d); {逆向边}
       i:=b[i];
    end;
inc(sum,d); {总流量增加d}
```

主程序：

```
for i:=1 to n do b[i]:= -1;  {初始化增广路径}
b[1]:=0;
while findflow(1) do {增广流}
    begin
        addflow;
        for i:=1 to n do b[i]:=-1;
        b[1]:=0;
    end;
writeln(sum);          {输出最大流}
for i:=1 to n do       {输出每条边的流量}
  for j:=1 to n do
    if f[i,j]>0 then writeln(i,'-->',j,' ',f[i,j]);
```

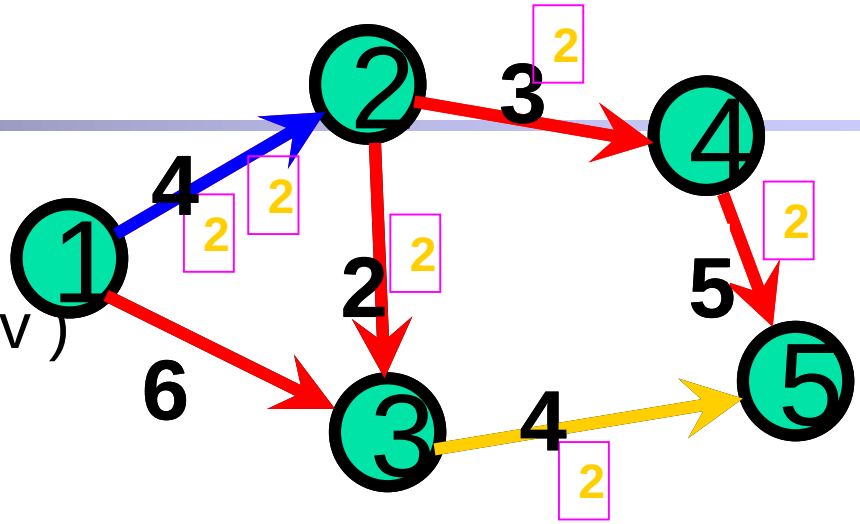
优化

- 残留网络 d 的设置：

  若存在 (u, v) 则

  d (u, v) = c (u, v) – f (u, v)

  d (v, u) = f (u, v)



d (u, v) 是 从 u 到 v 能增加的最大流量

理解:

(u,v) 的流量为f(u,v),作为正向边还可以增加的量是c (u, v) – f (u, v)，作为逆向边，还可以增加的流量为： f (u, v)。

增广路上正向边流量增加，逆向边增加流量减少。

d (u, v) = c (u, v)

d (v, u) = 0

深搜找增广路径过程

```
function find( k:integer ):boolean;
        if  k=n  then  exit(true);
        for  i:=1  to  n  do
                if  (b[i]= -1)  and  (d[k,i]>0)  then
                        [ b[i]:=k;
                         if  find(i)  then  exit(true);
                                        // 此处b[i]不需要变回-1]
        exit(false);
// b[1]=0; b[2~ n]= -1; 主函数中调用find(1)
```

广搜找增广路径过程

```
function bfsbfs:boolean;                a 是广搜队列
    for i:=1 to n do  b[i]:=-1;        b 是前驱
    b[1]:=0;  a[1]:=1;  open:=0;   closed:=1;
    while  open<closed  do
        [ inc(open); k:=a[open];
         for i:=1 to n do                d 是残余流量
             if (b[i]= -1) and (d[k,i]>0) then
                 [inc(closed); a[closed]:=i; b[i]:=k;
                 if i=n then exit(true);   找到增广路]
        ]
   exit(false);                          没找到增广路
```

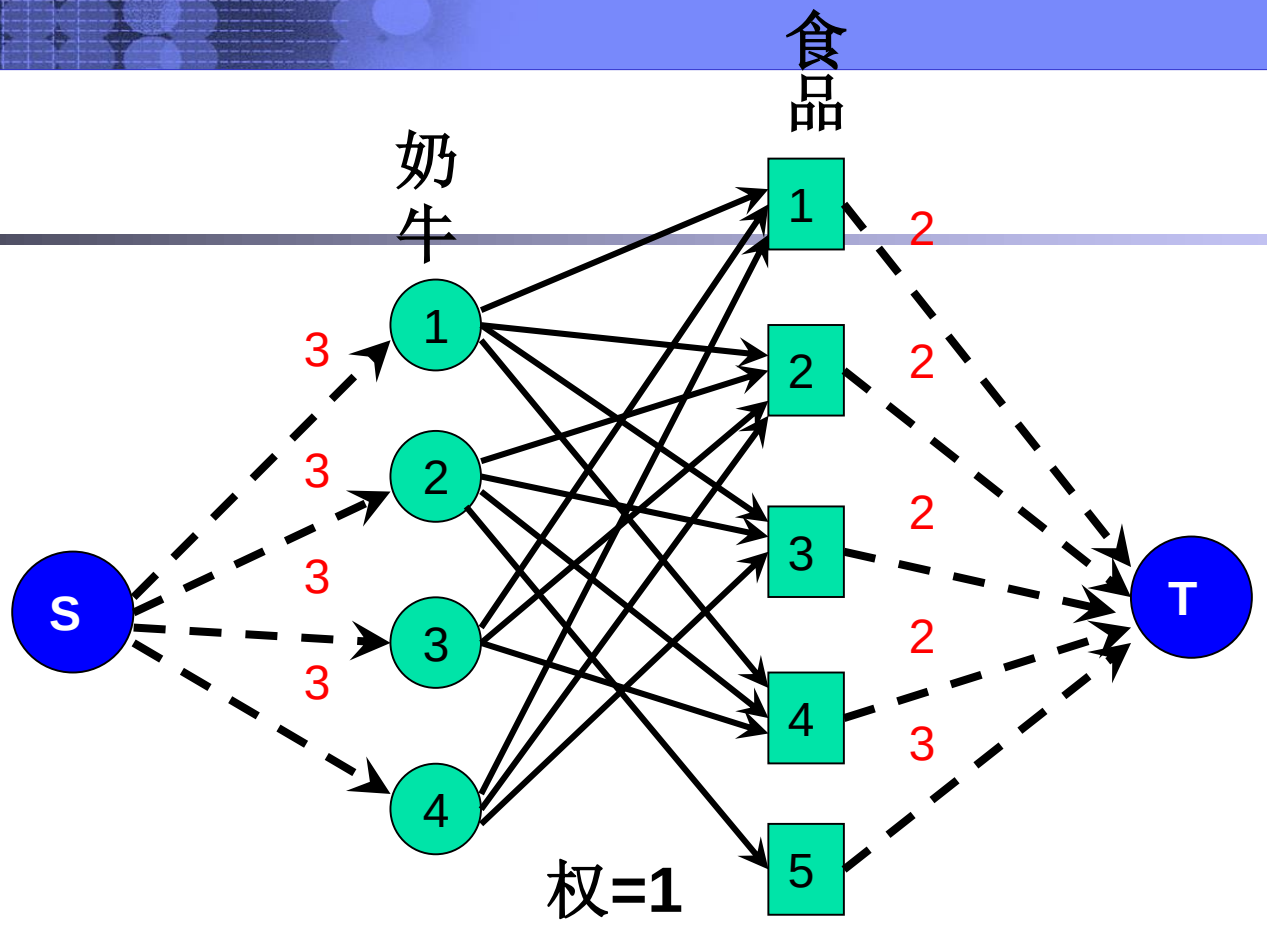
增广过程

```
min:=maxint;
i:=n;
while b[i]<>0 (i<>1) do //逆向求增加流min
      [ if min>d[b[i],i] then  min:=d[b[i],i];
       i:=b[i];]
i:=n;
while b[i]<>0 (i<>1) do// //逆向修改流量
      [ dec(d[b[i],i],min);  inc(d[i,b[i]],min);
       i:=b[i];]
inc(sum,min);          sum是总流量
```

问题：奶牛的新年晚会

奶牛们要举办一次别开生面的新年晚会。每头奶牛会做一些不同样式的食品（单位是盘）。到时候他们会把自己最拿手的不超过k样食品各做一盘带到晚会，和其他奶牛一起分享。但考虑到食品太多会浪费掉，他们给每种食品的总盘数都规定了一个不一定相同的上限值。这让他们很伤脑筋，究竟应该怎样做，才能让晚会上食品的总盘数尽量的多呢？

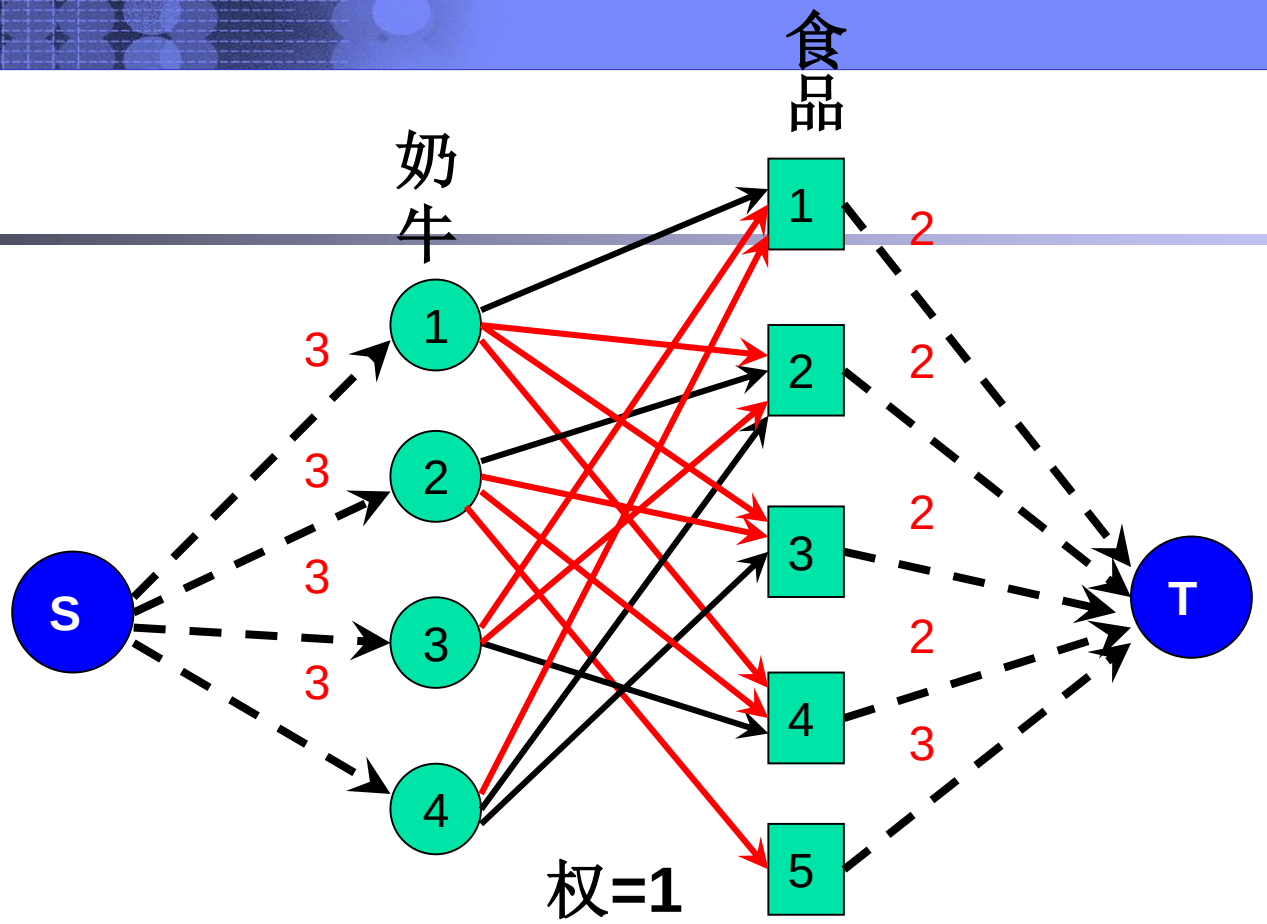
例如：有4头奶牛，每头奶牛最多可以带3盘食品。一共有5种食品，它们的数量上限是2、2、2、2、3。奶牛1会做食品1…4，奶牛2会做食品2…5，奶牛3会做食品1、2、4，奶牛4会做食品1…3。那么最多可以带9盘食品到晚会上。即奶牛1做食品2…4，奶牛2做食品3…5，奶奶3做食品1、2，奶牛4做食品1。这样，4种食品各有2、2、2、2、1盘。



边：S→奶牛, 保证每头奶牛带的食品的最大量。

边：食品→T, 保证每种食品的最大数量。

食品的总盘数最大值=S到T的最大流



S到T的最大流=9

数学建模——模型的选择、关系的简化

- 很多问题都是通过建立图论模型解决的
- 图论中常见的模型有序列、树、各种图
- 如何有效选择数学模型，简化原问题中元素之间的关系是数学建模的关键

建模步骤的总结

- 模型的准备
  - 提出问题，搜集数据。
- 模型的假设
  - 根据实际情况，提出合理的假设简化问题。
- 模型的建立
  - 根据所作的假设分析对象的因果关系，利用对象的内在规律和适当的数学工具，构造各个量间的等式关系或其它数学结构。