算法概论

第六讲：图算法

薛健

Last Modified: 2019.12.29

主要内容

1 基本概念 **1**

2 图的表示和数据结构 **2**

3 图的搜索与遍历 **4**

3.1 深度和广度优先搜索 4
3.2 有向图的深度优先搜索 6
3.3 有向无环图 10
3.4 有向图的强连通分量 12
3.5 无向图的深度优先搜索 15

4 最小生成树问题 **18**

4.1 基本问题 18
4.2 Prim 算法 19
4.3 Kruskal 算法 23

5 单源最短路径问题 **25**

5.1 基本问题 25
5.2 Dijkstra 算法 26

1 基本概念

图问题

- 不仅在计算机相关领域，在现实生活中的其他各个领域内都有大量的问题可以抽象为某种图的问题，因此基本的图算法应用范围十分广泛
- 所有与图相关的问题可以粗略地分为以下几类：
  1. **简单图问题**：已知线性复杂度 $(W(n) \in O(n))$ 的算法用于解决该类问题，最典型的就是图的遍历，这类算法可以解决规模非常大的问题
  2. **中等图问题**：已知多项式复杂度 $(W(n) \in O(n^a), a > 1)$ 的算法用于解决该类问题，这类算法可以解决规模相对较大的问题
  3. **困难图问题**：用于解决该类问题且复杂度低于多项式级别的算法还未找到，现有的算法仅能解决规模很小的问题 $\Rightarrow$ 算法研究的前沿

基本概念

Definition 1.1 (有向图 (directed graph / digraph))**. 有向图** $G = (V, E)$，其中 V 是一组顶点 (vertex) 也称结点 (node) 的集合，E 是一组顶点有序对的集合。E 中的顶点有序对称为**边** (edge) 或**有向边** (directed edge) 或**弧** (arc)。对于 E 中的有向边 (v, w)，v 称为尾 (tail)，w 称为头 (head)，在图中一般用箭头 $v \to w$ 表示

Definition 1.2 (无向图 (undirected graph))**. 无向图** $G = (V, E)$，其中 V 是一组顶点 (vertex) 也称结点 (node) 的集合，E 是一组顶点无序对的集合。E 中的顶点有序对称为**边** (edge) 或**无向边** (undirected edge)。E 中的每一条边都是 V 的包含 2 个元素的子集。E 中的边 $\{v, w\}$，在图中一般用连接两个顶点的直线 $v - w$ 表示。

Definition 1.3 (子图、对称图和完全图)**.**

- **子图** (subgraph)：$G = (V, E)$ 的子图是满足 $V' \subseteq V, E' \subseteq E$ 且 $E' \subseteq V' \times V'$ 的图 $G' = (V', E')$；
- **对称图** (symmetric graph)：是有向图，且其中的每一条边都存在一条与其方向相反的边；
- **完全图** (complete graph)：每两个顶点之间有边相连的图，一般是指无向图

Definition 1.4 (邻接关系 (adjacency relation))**.** 由有向图或无向图 $G = (V, E)$ 的边可以在顶点集合 V 上定义二元关系 A，对于 $v, w \in V$，vAw 当且仅当 $vw \in E$，该二元关系称为 **邻接关系**。若 G 是无向图，则 A 是对称的。

Definition 1.5 (路径 (path))**.** 图 $G=(V,E)$ 由顶点 v 到顶点 w 的**路径**是 E 中的一组边构成的序列 $v_0v_1, v_1v_2, \cdots, v_{k-1}v_k$，其中 $v=v_0, w=v_k$。该路径的长度为 k。一个独立的顶点 v 可看作从 v 到其自身的长度为 0 的路径。$v_0, v_1, \cdots, v_k$ 各不相等的路径称为**简单路径** (simple path)。若顶点 v 和 w 之间存在一条路径，则称顶点 w 对顶点 v **可达** (reachable)。

Definition 1.6 (连通和强连通)**.**

- 称无向图 $G=(V,E)$ 是**连通的** (connected)，当且仅当 V 中的任意一对顶点 v 和 w 之间存在一条路径；
- 称有向图 $G=(V,E)$ 是**强连通的** (strongly connected)，当且仅当 V 中的任意一对顶点 v 和 w 之间存在一条从 v 到 w 的路径；

Definition 1.7 (环路 (cycle))**.**

- 有向图中的**环路**是指一条首顶点和末顶点相同的非空路径，若环路中除首末顶点外其他顶点各不相同，则该环路称为**简单环路** (simple cycle)；
- 无向图中的**环路**定义与有向图类似，额外条件是环路中任意出现多次的边必须为同一朝向，即：若环路 $vv_1, v_1v_2, \cdots, v_kv$ 中存在 $v_i=x, v_{i+1}=y$，则不允许再出现 $v_j=y, v_{j+1}=x$；
- 不存在环路的图称为**无环图** (acyclic graph)；
- 无向无环图称为**无向森林** (undirected forest)，若该图同时是连通的，则称为**自由树** (free tree) 或**无向树** (undirected tree)；
- 有向无环图通常缩写为 DAG

Definition 1.8 (连通分量和强连通分量)**.**

- 无向图 $G=(V,E)$ 的**连通分量** (connected component) 是指 G 的最大连通子图；
- 有向图 $G=(V,E)$ 的**强连通分量** (strongly connected component) 是指 G 的最大强连通子图；
- “最大”的含义：不是任何其他连通（强连通）子图的子图

Definition 1.9 (带权图 (weighted graph))**.** **带权图**是一个三元组 (V,E,W)，其中 (V,E) 是图，W 是从 E 到 $\mathbf{R}$ 的函数，对于 E 中的边 e，$W(e)$ 称为 e 的权。

2 图的表示和数据结构

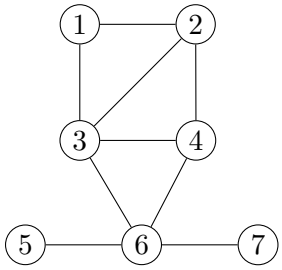
图的矩阵表示和链表表示

- **邻接矩阵表示**：用一个 $n\times n$ 的矩阵 $A=(a_{ij})$ 表示图 G，n 为顶点数

  - 无向图：$a_{ij} = \begin{cases} 1 & \text{if } v_iv_j \in E \\ 0 & \text{otherwise} \end{cases}$
  - 有向带权图：$a_{ij} = \begin{cases} W(v_iv_j) & \text{if } v_iv_j \in E \\ c & otherwise \end{cases}$
  - 其中 c 是一个常数，根据具体的应用环境而有所不同，例如权值代表某种开销的时候，一般取 $c = \infty$
  - 对无向图来说，其邻接矩阵是一个对称阵

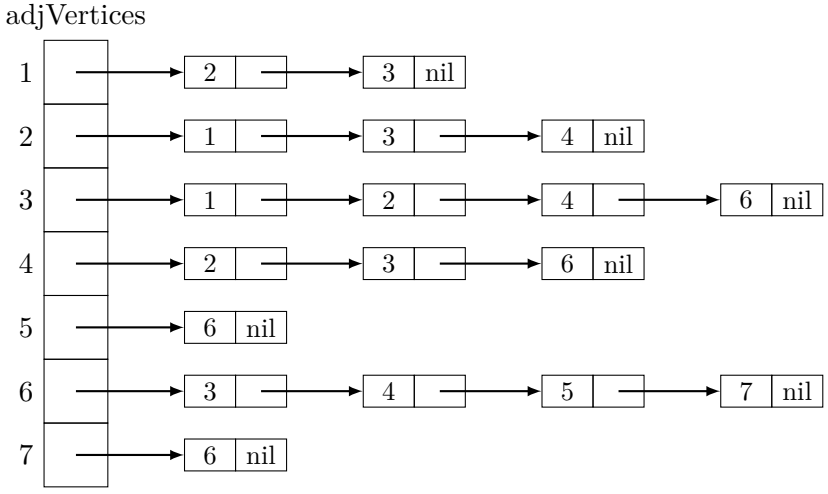
- **邻接链表表示**：为图 G 的每个顶点生成一个链表，其中存储从该顶点出发的所有边的信息（结束顶点、权值等）
  - 对无向图来说，每条边被记录了两次
  - 对有向图来说，每条边只被记录一次

无向图的邻接矩阵表示

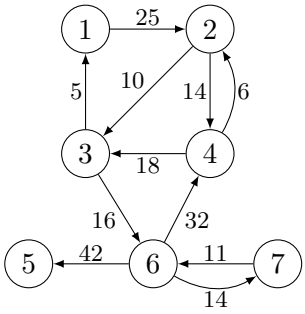


$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

无向图的邻接链表表示

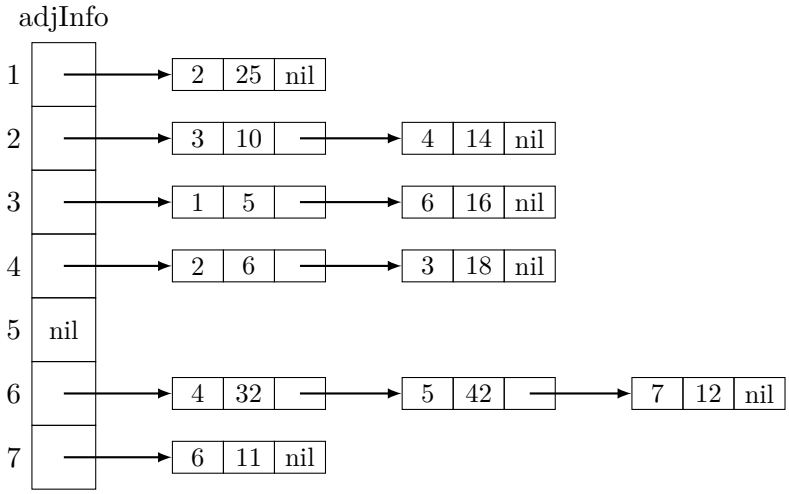


有向图的邻接矩阵表示



$$\begin{bmatrix} 0 & 25 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10 & 14 & \infty & \infty & \infty \\ 5 & \infty & 0 & \infty & \infty & 16 & \infty \\ \infty & 6 & 18 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32 & 42 & 0 & 14 \\ \infty & \infty & \infty & \infty & \infty & 11 & 0 \end{bmatrix}$$

有向图的邻接链表表示



3 图的搜索与遍历

3.1 深度和广度优先搜索

深度优先搜索 (Depth-First Search)

- 图的深度优先搜索（或遍历）是一般的树遍历算法的推广
- 其要点是:不断向前探索 (explore),无路可走的时候原路返回 (backtrack)
- 深度优先搜索算法概要:

Algorithm DFS(G, v)

```
1 标记 v 为“发现”;
2 foreach G 中与 v 有边相连的顶点 w do
3     if w 状态为“未发现” then
4         DFS(G, w);
5     else
6         “检查”边 vw 而不访问 w;
7     end
8 end
9 标记 v 为“结束”;
```

所有顶点的遍历

- 对于给定的图，从某一顶点出发做深度优先搜索并不是所有其他顶点都能到达

- 如果要遍历图 G 的所有顶点，则在深度优先搜索的基础上做一些额外的工作

Algorithm DFSSweep(G)

```
1 将 G 中所有顶点标记为“未发现”;
2 foreach v ∈ G do
3     if v 的状态为“未发现” then
4         DFS(v);
5     end
6 end
```

用深度优先搜索求连通分量

Algorithm ConnectedComponents$(adjVertices[], n)$

```
1 初始化 color[1], ··· , color[n] 为 white;
2 for v ← 1 to n do
3     if color[v] = white then CCDFS(adjVertices, color, v, v, cc) ;
4 end
5 return cc;
```

Procedure CCDFS($adjVertices[], color[], v, ccNum, cc[]$)

```
1  color[v] ← gray; cc[v] ← ccNum; remAdj ← adjVertices[v];
2  while remAdj ≠ null do
3  |   w ← First(remAdj);
4  |   if color[w] = white then
   |    CCDFS(adjVertices, color, w, ccNum, cc) ;
5  |   remAdj ← Rest(remAdj);
6  end
7  color[v] ← black;
```

时间复杂度：$\Theta(n+m)$； 空间复杂度：$\Theta(n)$

广度优先搜索 (Breadth-First Search)

- 广度优先搜索和深度优先搜索的“勇往直前”不同，它更像是有很多人同时从某一顶点出发，分别沿从该顶点出发的每条边向前搜索
- 一个非递归的广度优先搜索算法：

Algorithm BreadthFirstSearch($adjVertices[], n, s$)

```
1  初始化 color[1], ··· , color[n] 为 white;
2  parent[s] ← −1; color[s] ← gray;;
3  Queue pending ← Create(n); Enqueue(pending, s);
4  while pending 非空 do
5  |   v ← Front(pending); Dequeue(pending);
6  |   foreach 链表 adjVertices[v] 中的顶点 w do
7  |   |   if color[w] =white then
8  |   |   |   color[w] ← gray; Enqueue(pending, w); parent[w] ← v;
9  |   |   end
10 |   end
11 |   color[v] ← black;
12 end
13 return parent;
```

3.2 有向图的深度优先搜索

深度优先搜索树 (Depth-First Search Tree)

Definition 3.1 (深度优先搜索树和森林). 在深度优先搜索过程中连接处于“未发现”状态的顶点的边构成一棵以出发顶点为根的树，称为**深度优先搜索树**，或称**深度优先生成树** (depth-first spanning tree)。若从出发顶点不能到达图 G 中所有顶点，则对图 G 的完全深度优先遍历将把顶点分割成几棵互不相交的深度优先搜索树，称为**深度优先搜索森林** (depth-first search forest)，或称**深度优先生成森林** (depth-first spanning forest)。

Definition 3.2 (祖先 (ancestor))**.** 在图 G 的深度优先搜索树中，若顶点 v 在从根到顶点 w 的路径上，则称顶点 v 是顶点 w 的**祖先**，若同时满足 $v \neq w$，则称 v 是 w 的**严格祖先** (proper ancestor)，相应的，称 w 为 v 的**子孙** (descendant)。

有向图边的分类

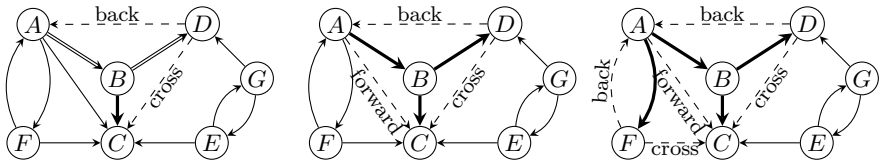
Definition 3.3. 有向图 G 的边可以根据它们被访问的方式分成如下几类：

1. 若当 vw 被访问时 w 状态为“未发现”，则称 vw 为**树边** (tree edge)，v 成为 w 的父结点；

2. 若 w 是 v 的祖先，则称 vw 为**返回边** (back edge) (包括 vv)；

3. 若 w 是 v 的子孙，但在访问 vw 时，w 已经被访问过（即状态为“结束”），则称 vw 为**子孙边** (descendant edge) (或称**前向边** (forward edge))；

4. 若 w 和 v 之间无祖先或子孙关系，则称 vw 为**交叉边** (cross edge)



深度优先搜索算法框架

Algorithm DFSSweep($adjVertices[], n, \cdots$)

```
1 将状态颜色数组 color 中的元素初始化为 white;
2 for v ← 1 to n do
3     if color[v] = white then
4         vAns ← DFS(adjVertices, color, v, ···);
5         处理 vAns;
6     end
7 end
8 return ans;
```

DFS 算法框架

Algorithm DFS($adjVertices[], color[], v, \cdots$)

```
1  color[v] ← gray;
2  处理顶点 v (preorder processing);
3  remAdj ← adjVertices[v];
4  while remAdj ≠ null do
5      w ← First(remAdj);
6      if color[w] = white then
7          处理边 vw (tree edge);
8          wAns ← DFS(adjVertices, color, w, ···);
9          回溯处理边 vw (使用 wAns);
10     else 检查边 vw (nontree edge) ;
11     remAdj ← Rest(remAdj);
12 end
13 处理顶点 v (postorder processing)，计算结果 ans;
14 color[v] ← black;
15 return ans;
```

深度优先搜索结构

- 在一些深度优先搜索的应用中，经常需要知道这样一些信息：
  - 从深度优先搜索树的根到当前访问顶点的路径上有哪些顶点（已经被访问过的灰色顶点）；
  - 顶点第一次被访问和最后一次被访问的时间顺序关系
- 这些信息可以通过在搜索过程中将当前时间记录在 $discoverTime$ 和 $finishTime$ 两个数组中得到：
  - 顶点 v 颜色为 white：顶点尚未被发现
  - 当顶点 v 颜色变为 gray：当前时间记录在 $discoverTime[v]$
  - 当顶点 v 颜色变为 black：当前时间记录在 $finishTime[v]$
  - 若时间用递增的整数表示，则顶点 v 的活动区间 (active interval) 为
    $$active(v) = discoverTime[v], \cdots, finishTime[v]$$
  - 在顶点 v 的活动区间中，其颜色为 gray

深度优先搜索跟踪

- 对深度优先搜索算法框架稍作修改即可得到深度优先搜索跟踪算法，用于计算 $discoverTime$ 和 $finishTime$ 数组

Algorithm DFSTrace($adjVertices[], n$)

```
1 将 color 中的元素初始化为 white;
2 global time ← 0;
3 global discoverTime, finishTime, parent;
4 for v ← 1 to n do
5     if color[v] = white then
6         parent[v] ← −1;
7         TDFS(adjVertices, color, v);
8     end
9 end
```

TDFS

Procedure TDFS($adjVertices[], color[], v$)

```
1  color[v] ← gray;
2  time ← time + 1; discoverTime[v] ← time;
3  remAdj ← adjVertices[v];
4  while remAdj ≠ null do
5      w ← First(remAdj);
6      if color[w] = white then
7          parent[w] ← v;
8          TDFS(adjVertices, color, w);
9      end
10     remAdj ← Rest(remAdj);
11 end
12 time ← time + 1; finishTime[v] ← time;
13 color[v] ← black;
```

一些结论

Theorem 3.4. 根据前面对 $active(v)$ 和边分类的定义，算法 *DFSTrace* 作用于有向图 $G = (V, E)$ 后，对于任意顶点 $v \in V$ 和 $w \in V$，有如下结论成立：

1. 在 *DFS* 森林中，w 是 v 的子孙当且仅当 $active(w) \subseteq active(v)$，若 $v \neq w$，则 $active(w) \subset active(v)$
2. 若在 *DFS* 森林中，w 和 v 无祖先/子孙关系，则它们的活动区间 $active(v)$ 和 $active(w)$ 不相交
3. 若边 $vw \in E$，则：
   (a) vw 是交叉边当且仅当 $active(w)$ 整个区间在 $active(v)$ 之前
   (b) vw 是子孙边当且仅当存在第三个顶点 x，使得 $active(w) \subset active(x) \subset active(v)$

(c) vw 是树边当且仅当 $active(w) \subset active(v)$，并且不存在第三个顶点 x，使得 $active(w) \subset active(x) \subset active(v)$

(d) vw 是返回边当且仅当 $active(v) \subset active(w)$

3.3 有向无环图

基本概念

- 有向无环图 (DAG) 是一种非常重要的有向图：
  - 很多实际问题可以用 DAG 来描述，例如任务调度问题 (scheduling problem)，某项任务的开始依赖于其他任务的结束，如果任务的依赖关系中存在环路，则意味着发生了死锁 (deadlock)
  - 很多实际问题用 DAG 解决比用一般有向图解决来得更简单和有效，很多时候可以使时间复杂度从指数级别降到线性级别
  - 在数学上，一个 DAG 对应于顶点间的某种偏序关系 (partial order relation)
- 对于给定 DAG，可以给出其顶点的拓扑顺序 (topological order)：

  Definition 3.5 (拓扑顺序). $G = (V, E)$ 是一个包含 n 个顶点的有向图，图 G 的**拓扑顺序**就是给 V 中的顶点赋予 **拓扑数** (topological number) $1, \cdots, n$ 来表示顶点顺序，使得对每一条边 $vw \in E$，顶点 v 的拓扑数小于顶点 w 的拓扑数；若顶点 v 的拓扑数大于顶点 w 的拓扑数则称为 **逆拓扑顺序** (reverse topological order)。

逆拓扑排序

Lemma 3.6. 若有向图 G 中存在环路，则 G 无拓扑顺序。

- 拓扑排序是 DAG 的基本问题，任意给定 DAG 至少有一种拓扑顺序
- 若解决了拓扑排序问题，很多实际问题便迎刃而解（recall：递归算法的子问题图）
- 通过简单修改前面给出的深度优先搜索算法框架便可以得到一个逆拓扑排序算法：

Algorithm `ReverseTopoOrdering`($adjVertices[], n$)

```
1 将 color 中的元素初始化为 white; topo ← new int[n + 1];
2 global topoNum ← 0;
3 for v ← 1 to n do
4 |   if color[v] = white then RTODFS(adjVertices, color, v, topo) ;
5 end
6 return topo;
```

RTODFS

Procedure RTODFS($adjVertices[], color[], v, topo[]$)

```
1  color[v] ← gray;
2  remAdj ← adjVertices[v];
3  while remAdj ≠ null do
4  |   w ← First(remAdj);
5  |   if color[w] = white then RTODFS(adjVertices, color, w, topo) ;
6  |   remAdj ← Rest(remAdj);
7  end
8  topoNum ← topoNum + 1; topo[v] ← topoNum;
9  color[v] ← black;
10 return ans;
```

Theorem 3.7. 图 G 是给定包含 n 个顶点的 DAG,算法 ReverseTopoOrdering 在返回的 $topo$ 数组中给出了 G 的一个逆拓扑顺序，因而任意一个 DAG 至少有一个逆拓扑顺序和一个拓扑顺序。

证明要点

1. RTODFS 访问每个顶点一次（状态从 gray 到 black），第 8 行执行了 n 次，则数组 $topo$ 中保存了从 1 到 n 不同整数；
2. 验证 $\forall vw \in E,\ topo[v] > topo[w]$：
   - vw 不可能是返回边，否则表明图中存在环路，与 DAG 矛盾；
   - vw 是除返回边以外的其他类型边，则当 v 颜色变为 $black$ 时，w 颜色已经变成 $black$，即 $topo[w]$ 赋值早于 $topo[v]$，而 $topoNum$ 只增不减，因此 $topo[v] > topo[w]$。

关键路径 (Critical Path) 分析

- 关键路径分析是与拓扑排序有关的优化问题，其优化目标是找到给定 DAG 中的最长路径

Definition 3.8 (任务调度问题). 设某项目由 n 个任务组成，分别标记为 $1, \cdots, n$，每一个任务都有一个依赖关系列表，表明该任务的开始直接依赖于哪些任务的结束，则：

- 某个任务 v 的**最早开始时间** (earliest start time, est) 是指：
  1. v 与其他任务无依赖关系：$est = 0$
  2. v 与其他任务有依赖关系：est 是其所有依赖任务最早结束时间的最大值
- 某个任务的**最早结束时间** (earliest finish time, eft) 是其最早开始时间加上其任务本身的执行时间

- 该项目的**关键路径**是一个任务序列 $v_0, v_1, \cdots, v_k$，且满足：
  1. v_0 与其他任务无依赖关系
  2. 每个 v_i 直接依赖于 v_{i-1}，即 v_i 的 est 等于 v_{i-1} 的 eft
  3. v_k 的最早结束时间是项目的所有任务最早结束时间中的最大值

求关键路径

- 关键路径的意义：若要提前完成项目，必须缩短关键路径上任务的执行时间，仅缩短非关键路径任务的执行时间对整个项目来说是没有价值的
- 同样通过修改 DFS 算法框架可以得到关键路径算法：

Algorithm `CriticalPath`$(adjVertices[], n, duration[])$

```
1 将 color 中的元素初始化为 white;
2 global critDep, eft;
3 for v ← 1 to n do
4 |   if color[v] = white then
5 |   |   CPDFS(adjVertices, color, v, duration);
6 |   end
7 end
```

`CPDFS`

Procedure `CPDFS`$(adjVertices[], color[], v, duration[])$

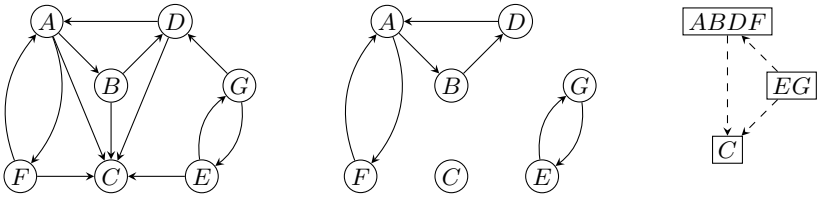
```
1  color[v] ← gray;
2  est ← 0; critDep[v] ← −1;
3  remAdj ← adjVertices[v];
4  while remAdj ≠ null do
5  |   w ← First(remAdj);
6  |   if color[w] = white then
7  |   |   CPDFS(adjVertices, color, w, duration);
8  |   end
9  |   if eft[w] ≥ est then  est ← eft[w]; critDep[v] ← w ;
10 |   remAdj ← Rest(remAdj);
11 end
12 eft[v] ← est + duration[v];
13 color[v] ← black;
```

3.4 有向图的强连通分量

有向图的凝聚图 (Condensation Graph)

Definition 3.9 (凝聚图). 设 $S_1, S_2, \cdots, S_p$ 是有向图 $G = (V, E)$ 的强连通分量，则 G 的**凝聚图**是 有向无环图 $G\downarrow = (V', E')$，其中 V' 有 p 个顶点 $s_1, \cdots, s_p$，$s_i s_j \in E'$ 当且仅当 $i \neq j$ 且在 E 中存在从 S_i 中的某一顶点到 S_j 中某一顶点的边，即将强连通分量 S_i 凝聚为一个顶点 s_i。

Example 3.1.



强连通分量的性质

Definition 3.10 (转置图 (transpose graph)). 将有向图 G 的所有边方向反转得到的图称为 G 的**转置图**，用 G^T 表示。显然，$(G\downarrow)^T = (G^T)\downarrow$。

Definition 3.11 (强连通分量的领头顶点 (leader)). 给定有向图 G 的强连通分量 S_i，$i = 1, \cdots, p$，对 G 进行深度优先搜索的过程中所发现的属于 S_i 的第一个顶点称为强连通分量 S_i 的**领头顶点**，记作 v_i。

Lemma 3.12. 有向图 G 的深度优先搜索森林中的每一棵树均包含一个或多个完整的强连通分量，且不包含“部分”强连通分量。

Corollary 3.13. 在深度优先搜索过程中，强连通分量 S_i 的所有顶点中领头顶点 v_i 最后一个结束（颜色变为 *black*）。

Lemma 3.14. 在深度优先搜索过程中，当某个领头顶点 v_i 被发现时，不存在从 v_i 到其他灰色（*gray*）顶点的路径。

Lemma 3.15. 若 v 是强连通分量 S 的领头顶点，x 是另外一个强连通分量中的顶点，并且存在一条从 v 到 x 的路径，则在深度优先搜索过程中，当 v 被发现时，x 要么是黑色（*black*）顶点，要么存在一条从 v 到 x 的由白色（*white*）顶点（包括 x）构成的路径，且在任何情况下 v 比 x 更晚结束

强连通分量算法

- 深度优先搜索可以用于求有向图 G 的所有强连通分量，具体算法可分为如下两个阶段：

  1. 对 G 进行标准深度优先搜索，在每个顶点的结束时间将其压入堆栈
  2. 对图 G 的转置图 G^T 进行深度优先搜索，但在主循环中根据退栈顺序开始某个顶点 (white) 的搜索，而不再根据 $adjVertices$ 数组中的自然顺序；在搜索过程中，将每个顶点所属的强连通分量的领头顶点存储在数组 scc 中

- 算法描述 ⇒ 课后练习
- 算法的正确性：

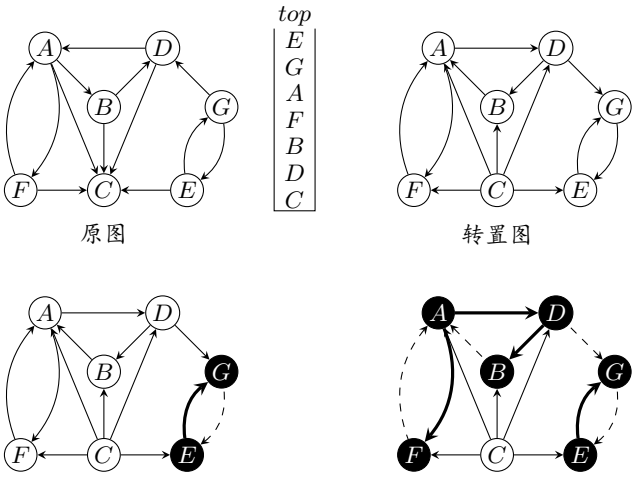
  Lemma 3.16. *在上述算法的阶段 2，每次从栈中弹出的白色（white）顶点一定是阶段 1 中某个强连通分量的领头顶点。*

强连通分量算法 (cont.)

Theorem 3.17. *在上述算法的阶段 2，每一棵深度优先搜索树包含且仅包含一个强连通分量的所有顶点。*

Proof. 由前面的引理可知每棵深度优先搜索树包含一个或多个强连通分量，且不包含部分强连通分量，因此我们只要证明在阶段 2 中每棵深度优先搜索树仅包含一个强连通分量。假设 v_i 是阶段 1 中强连通分量 S_i 的领头顶点，则根据引理，它是 S_i 中最后被压入堆栈的顶点，则在阶段 2 中当 v_i 从栈中弹出为白色顶点时，v_i 是阶段 2 深度优先搜索树的根，假设有另外一个强连通分量 S_j 其领头顶点为 v_j，在 G^T 中存在一条路径从 v_i 到达 S_j 中某一顶点，即可到达 v_j，则在图 G 中存在 v_j 到 v_i 的路径，根据前面的引理，v_j 在阶段 1 中比 v_i 更晚结束，因此在阶段 2 中更早从栈中弹出，当 v_i 被弹出时，S_j 中所有顶点均已变为 black，所以从 v_i 开始的搜索不可能再脱离强连通分量 S_i。 □

例子



课后习题

Exercise (8). 给定有向图 G：

(1) 证明图 G 的凝聚图 $G\downarrow$ 是有向无环图。

(2) 若图 G 以邻接表的形式存储，试写出一个算法求图 G 的转置图 G^T。

deadline: 2020.01.04

3.5 无向图的深度优先搜索

无向图的深度优先搜索算法框架

- 一般情况下，无向图的深度优先搜索可以采用有向图的搜索算法，实际上无向图在存储结构上等价于一个对称有向图（每条边裂为两条方向相反的边）
- 在某些应用环境下需要保证每条边在深度优先搜索过程中仅访问一次，在这种情况下需要忽略等价对称有向图搜索过程中出现的某些边，这些边都是对应无向图搜索中第二次遇到的边，包括：
  - 从 v 到 p 的返回边 (back edge)，其中 pv 为树边
  - 从 v 到 w 的前向边 (forward edge)，此时 w 必为 black
  - 在对称有向图的深度优先搜索中不可能出现交叉边 (cross edge)
- 算法框架 `UDFSSweep` 与 `DFSSweep` 基本相同。

`UDFS` 算法框架

Algorithm `UDFS`$(adjVertices[], color[], v, p, \cdots)$

```
1  color[v] ← gray;
2  处理顶点 v (preorder processing);
3  remAdj ← adjVertices[v];
4  while remAdj ≠ null do
5      w ← First(remAdj);
6      if color[w] = white then
7          处理边 vw (tree edge);
8          wAns ← UDFS(adjVertices, color, w, v, ···);
9          回溯处理边 vw (使用 wAns);
10     else if color[w] = gray and w ≠ p then 检查边 vw (back edge) ;
11     remAdj ← Rest(remAdj);
12 end
13 处理顶点 v (postorder processing)，计算结果 ans;
14 color[v] ← black;
15 return ans;
```

重连通分量

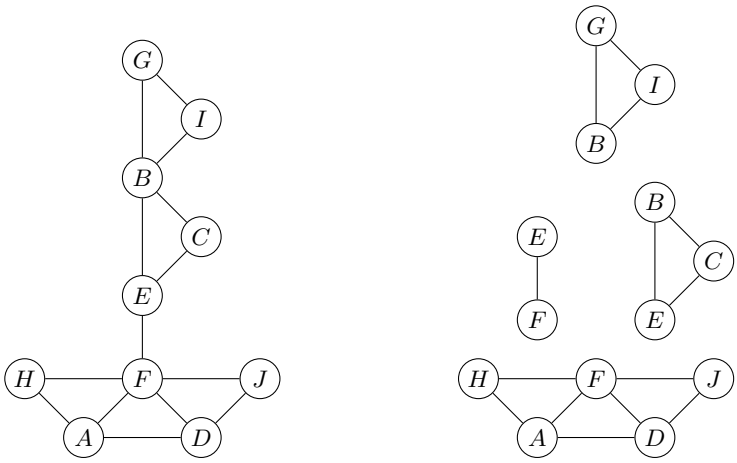
- 一些和无向图相关的实际问题：
  - 如果某个城市的机场由于天气原因而关闭，则其他任意两个城市之间是否仍有航线可以连接？
  - 若网络上某台路由器出现故障，该网络中任意两台计算机之间是否仍然可以通信？

- 问题的抽象：从一个给定连通图中去掉任一顶点 (包括和其连接的边)，剩余的子图是否仍然是连通图？

Definition 3.18 (重连通分量 (biconnected component))**.** 若从连通无向图 G 中移除任一顶点以及和该顶点连接的边后剩下的子图仍然是连通图，则称 G 是**重连通的** (biconnected)。无向图 G 的**重连通分量**是指 G 的最大重连通子图。

Definition 3.19 (关节点 (articulation point))**.** 若无向图 G 中存在两个不同顶点 w 和 x 之间的所有路径均通过不同于这两个顶点的第三个顶点 v，则称 v 为**关节点**。

例子



重连通分量算法设计

- 显然，一个无向连通图是重连通图当且仅当该图中不存在关节点

- 在深度优先搜索过程中，当从顶点 w 回溯到顶点 v 时，若以 w 为根的深度优先搜索子树中任一顶点都不存在到 v 的祖先的返回边 (back edge)，则意味着 v 一定在从深度优先搜索树的根到 w 的每一条路径上，因此 v 是一个关节点

- 以 w 为根的子树加上所有从其中出发的返回边以及边 vw 可以在顶点 v 与原图分离，但可能包含不止一个重连通分量，如果在搜索过程中一旦发现关节点即分离重连通分量，则可保证分离顺序由叶结点向根结点接近，从而使每次分离的都是单个的重连通分量

- 可以在深度优先搜索的递归过程中计算并返回一个返回时间 $back$，若从当前顶点出发通过递归调用搜索的子树返回的返回时间不小于当前顶点的发现时间 ($discoverTime[v]$)，则表明当前顶点是一个关节点

算法正确性保证

Theorem 3.20. 在深度优先搜索树中，非根顶点 v 是关节点当且仅当 v 不是叶结点并且存在某棵 v 的子树没有返回边连接到 v 的祖先结点（不包括 v 本身）。

Proof. (1) 若 v 为关节点，则存在其他两个不同顶点 x 和 y，连接 x 和 y 的每条路径都经过 v，则 x 和 y 中至少有一个是 v 的子孙，否则 x 和 y 之间一定存在某条路径不经过 v，因此 v 不是叶结点。下面假设 v 的每一棵子树都有一条连接到 v 的祖先的返回边，有两种情况：x 和 y 中只有一个是 v 的子孙，则该子孙结点可通过返回边跳过 v 到达另一个顶点，与前提矛盾；x 和 y 都是 v 的子孙，则二者不可能在同一棵子树中，而在不同子树中则可以通过各自的返回边跳过 v 相互连接，同样与前提矛盾。因此肯定存在某棵子树没有返回边连接到 v 的祖先。(2) 若 v 不是叶结点并且存在某棵 v 的子树没有返回边连接到 v 的祖先结点，则取 x 为 v 的某个祖先，y 为没有通向 v 祖先结点的返回边的子树中的某一顶点，则显然 x 和 y 之间的每条路径都经过 v。 □

重连通分量算法

- 根据上述分析，通过修改无向图的深度优先搜索算法框架可以得到重连通分量算法

- 在分离重连通分量时，所有边的输出可以通过一个栈结构来实现

Algorithm `Bicomponents`$(adjVertices[], n)$

```
1 global time ← 0;
2 global edgeStack ← CreateStack();
3 初始化 color[1], ··· , color[n] 为 white;
4 for v ← 1 to n do
5 |   if color[v] = white then
6 |   |   BicompDFS(adjVertices, color, v, −1);
7 |   end
8 end
```

`BicompDFS`

Algorithm `BicompDFS`$(adjVertices[], color[], v, p)$

```
1  color[v] ← gray; time ← time + 1; back ← discoverTime[v] ← time;
2  remAdj ← adjVertices[v];
3  while remAdj ≠ null do
4      w ← First(remAdj);
5      if color[w] = white then
6          Push(edgeStack, vw);
7          wBack ← BicompDFS(adjVertices, color, w, v);
8          if wBack ≥ discoverTime[v] then
9              从 edgeStack 中弹出边直至弹出 vw，输出一个重连通分量;
10         end
11         back ← min(back, wBack);
12     else if color[w] = gray and w ≠ p then
13         Push(edgeStack, vw);
14         back ← min(back, discoverTime[w]);
15     end
16     remAdj ← Rest(remAdj);
17 end
18 time ← time + 1; finishTime[v] ← time; color[v] ← black;
19 return back;
```

重连通分量算法分析

- 时间复杂度：
  - `Bicomponents` 的初始化工作开销为 $\Theta(n)$
  - 无向图的深度优先搜索算法框架复杂度为 $\Theta(n+m)$ (每个顶点处理 2 次，每条边处理 1 次)
  - 重连通分量的输出部分虽然每次开销可能不同，但总的来说每条边仅被压栈和退栈一次，因此总的开销为 $\Theta(m)$
  - 因此重连通分量算法的时间复杂度为 $\Theta(n+m)$
- 问题的推广：
  - k-连通分量：任意顶点之间存在 k 条不相交路径
  - 一个深度优先搜索求 3-连通分量的算法，参考：J. E. Hopcroft and R. E. Tarjan. Dividing A Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135–157, 1973

4 最小生成树问题

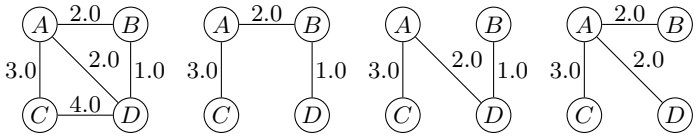
4.1 基本问题

问题描述

- 现实问题：现有一系列的车站、工厂、网络中的计算机等，如何用最小的代价将它们各自连接起来，使其两两之间都可到达？

Definition 4.1 (最小生成树 (minimum spanning tree))**.** 一个连通无向图 $G = (V, E)$ 的**生成树** (spanning tree) 是一棵树，并且是由 G 的所有顶点构成的子图；在带权图 $G = (V, E, W)$ 中，子图的权值是其所包含的所有边的权值之和，则带权连通无向图 G 的 **最小生成树** (MST) 就是图 G 权值最小的一棵生成树。

Example 4.1.



4.2 Prim 算法

基本思路

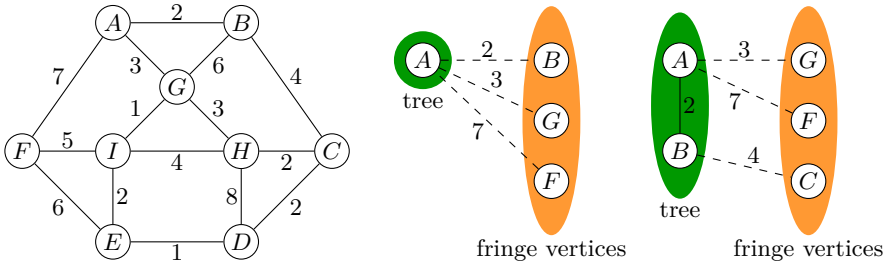
- 能否利用深度优先搜索？
- 贪心方法尝试：
  - 每次选择与当前子图相连（仅有一个顶点在子图中）的权值最小的边加入子图
  - 顶点类型：
    - tree　已经包含在构造的生成树中
    - fringe　不在树中，但与树中某一顶点相连
    - unseen　除上面两种情况外的其他顶点
- Prim **算法** (Robert C. Prim, 1957)：

Algorithm `PrimMST`(G, n)

```
1 初始化所有顶点为 unseen;
2 选择任一顶点 s 开始构造生成树，将其类型改为 tree;
3 将与 s 相连的所有顶点类型改为 fringe;
4 while 还有类型为 fringe 的顶点 do
5 |   从中选出一个所在边权值最小的顶点 v;
6 |   将其所在边加入生成树，并将 v 的类型修改为 tree;
7 |   将所有与 v 相连的类型为 unseen 的顶点类型改为 fringe;
8 end
```

算法演示

Example 4.2.



- 问题：
  - 算法的正确性：PrimMST 产生的是不是最小生成树？
  - 算法的有效性：PrimMST 的时间复杂度如何？（与实现相关）

算法的正确性

Definition 4.2 (最小生成树性质). 设 T 是带权无向连通图 $G = (V, E, W)$ 的一棵生成树，若将 G 中一条不在 T 中的边 uv 加入 T 则 T 中将出现一条环路，如果对任意一条这样的边 uv，其权值是产生的环路所包含的边中最大的，则称生成树 T 具有**最小生成树性质** (minimum spanning tree property)。

Lemma 4.3. 给定带权无向连通图 $G = (V, E, W)$，若 T_1 和 T_2 是 G 的两棵具有最小生成树性质的生成树，则 T_1 和 T_2 权值相等。

算法的正确性 (cont.)

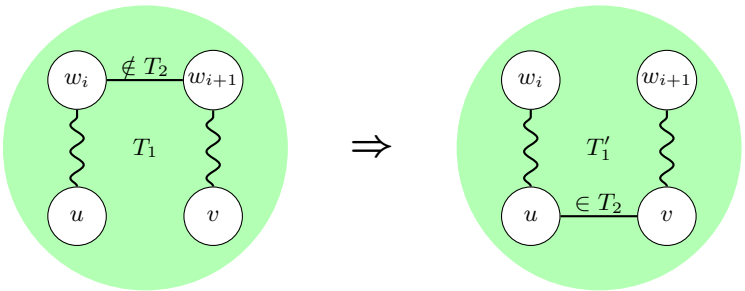
Proof. 用数学归纳法证明。对在 T_1 中而不在 T_2 中的边的数量 k 归纳。

奠基：$k = 0$，这时 T_1 和 T_2 完全相同，结论成立；

归纳：假设在 T_1 中而不在 T_2 中的边的数量 $< k$ 时结论成立，则边数 $= k$ 时，设 uv 是只在 T_1 或只在 T_2 的边中权值最小的边，不失一般性，假设 $uv \in T_2$，考虑 T_1 中从 u 到 v 的路径 $w_0, w_1, \cdots, w_p$，其中 $w_0 = u, w_p = v$ 且 $p \geq 2$，在这条路径中必然存在某条边不在 T_2 中，否则将在 T_2 中构成环路，设 $w_i w_{i+1} \notin T_2$，由于 T_1 具有最小生成树性质，因此 $W(w_i w_{i+1}) \leq W(uv)$，又 uv 是所有不同边中权值最小的边，因此 $W(w_i w_{i+1}) \geq W(uv)$，因此 $W(w_i w_{i+1}) = W(uv)$，将 uv 加入 T_1 同时将 $w_i w_{i+1}$ 移除，则可保证产生的新树 T_1' 仍是一棵生成树，且权值与 T_1 相等，而 T_1' 与 T_2 只相差 $k-1$ 条边，根据归纳假设它们的权值相等，因此 T_1 和 T_2 的权值也相等。 □

算法的正确性 (cont.)

- 示意图



算法的正确性 (cont.)

Theorem 4.4. 带权无向连通图 $G = (V, E, W)$ 的生成树 T 是最小生成树，当且仅当 T 具有最小生成树性质。

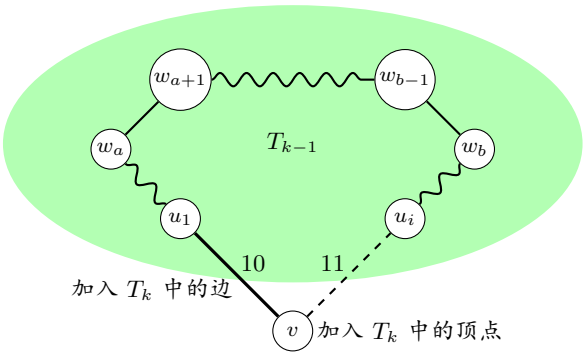
Lemma 4.5. 给定带权无向连通图 $G = (V, E, W)$, $|V| = n$，若 T_k 是 PrimMST 生成的包含 k 个顶点的树，$k = 1, \cdots, n$，G_k 是由这 k 个顶点及其相连的边构成的 G 的子图，则 T_k 在 G_k 中具有最小生成树性质。

Proof. **提示：** 用数学归纳法证明。在归纳步骤只需证明按 PrimMST 的步骤加入的边仍然能够保持新的树 T_k 具有最小生成树性质（反证法）。 □

Theorem 4.6. 算法 PrimMST 的输出是其输入图的最小生成树。

证明示意

- 在归纳步骤加入 T_{k-1} 的顶点为 v，加入的边为 vu_1，而 T_{k-1} 中其他在原图中与 v 有边相连的顶点为 $u_2, \cdots, u_d$
- 证明 T_k 具有最小生成树性质，即证明对于任意 $xy \in G_k$ 而 $xy \notin T_k$，xy 是加入 T_k 构成的环路中权值最大的边：
  1. $x \neq v, y \neq v$：$xy \in G_{k-1}$ 且 $xy \notin T_{k-1}$，根据归纳假设结论成立
  2. xy 是 $u_2v, \cdots, u_dv$ 中的一条边：反证法



算法的有效性保证

- 用最小优先队列来存储和维护 fringe 顶点集合：在优先队列中每个 fringe 顶点只需记录与 tree 顶点相连的边中权值最小的那一条，称为**候选边** (candidate edge)，该边权值即为优先队列元素的权值

Algorithm `PrimMST`(G, n)

```
1 初始化所有顶点为 unseen，优先队列 pq 为空;
2 选择任一顶点 s 设置其候选边为 (−1, s, 0);
3 Insert(pq, s, 0);
4 while pq 非空 do
5     v ← GetMin(pq);
6     DeleteMin(pq);
7     将 v 的候选边加入生成树，v 的类型修改为 tree;
8     UpdateFringe(pq, G, v);
9 end
```

`UpdateFringe`

Procedure `UpdateFringe`(pq, G, v)

```
1  foreach 与 v 相连的顶点 w do
2      newW ← W(v, w);
3      if w 类型为 unseen then
4          将 w 的候选边设置为 (v, w, newW);
5          w 的类型修改为 fringe;
6          Insert(pq, w, newW);
7      else if w 类型为 fringe and newW < GetPriority(pq, w) then
8          将 w 的候选边改为 (v, w, newW);
9          DecreaseKey(pq, w, newW);
10     end
11 end
```

$$T(n,m) \in O(nT(\texttt{GetMin}) + nT(\texttt{DeleteMin}) + mT(\texttt{DecreaseKey}))$$

Prim 算法时间复杂度分析

- 时间复杂度与优先队列的实现有关，但对一般的连通图总是有 $m > n$，因此总的来说 `DecreaseKey` 的复杂度更值得关注
- 当 Prim 设计这个算法时还没有出现优先队列的概念，因此只是简单地用数组实现
  - `GetMin` 和 `DeleteMin`: $\Theta(n)$
  - `DecreaseKey`: $\Theta(1)$
  - 因此 `PrimMST`: $\Theta(n^2)$
- 用二叉堆实现？$(\Theta(m \log n) \Rightarrow O(n^2 \log n))$
- 用配对森林实现：

- `DecreaseKey`: $\Theta(1)$
- `GetMin` 和 `DeleteMin` 在最坏情况下时间复杂度为 $\Theta(n)$，复杂度与简单数组实现类似，但是显然在平均情况下其时间复杂度要好
- 已知结果：two-pass pairing heaps，当 $m \in \Theta(n^{1+c}), c > 0$，`GetMin` 的分摊时间开销为 $\Theta(\log n)$，因此 `PrimMST` 复杂度为 $\Theta(m+n\log n) = \Theta(m)$
- 参考：Michael L. Fredman. On the Efficiency of Pairing Heaps and Related Data Structures. *Journal of the ACM* 46(4): 473–501, 1999

4.3 Kruskal 算法

基本思路

- 一个更“贪心”的方法：每次从剩余边集中取出权值最小且与生成树中已有边不构成环路的边加入到生成树中，直到所有的边都处理完毕
- Kruskal **算法** (Joseph B. Kruskal, 1956)：

Algorithm `KruskalMST`(G, n)

```
1 R ← E;
2 F ← ∅;
3 while R 非空 do
4 |   从 R 中移出权值最小的边 vw;
5 |   if vw 在 F 中不构成环路 then F ← F ∪ {vw} ;
6 end
7 return F;
```

算法的正确性

Definition 4.7 (生成树集). 对于带权无向图 $G = (V, E, W)$，其**生成树集** (spanning tree collection) 是其每个连通分量的生成树所构成的集合；**最小生成树集**是权值之和最小的生成树集，即每个连通分量的最小生成树的集合。

- `KruskalMST` 执行完毕后，G 中的每一个顶点都在某一棵树中吗？(孤立顶点)
- 若 G 无孤立顶点，则 `KruskalMST` 的输出是 G 的生成树集吗？即：是否 F 中的每一棵树恰是 G 的一个连通分量的生成树？

Lemma 4.8. 给定森林 F (即任意一个非连通无向无环图)，若边 $e = vw$ 不在 F 中，则 e 和 F 中的某些边构成环路当且仅当 v, w 在 F 的同一个连通分量中。

Proof. ‘$\Rightarrow$’：假设 e 和 F 中的某些边构成环路 $vw_1, \cdots, w_p w (p \geq 1)$，但 v, w 在 F 的不同连通分量 C_1 和 C_2 中，则去掉边 e 后，C_1 和 C_2 仍保持连通，这与 C_1 和 C_2 是不同连通分量矛盾；

‘$\Leftarrow$’：显然成立。 □

算法的正确性 (cont.)

- 若假设 G 中的某个连通分量对应 F 中两棵以上的树，则必存在 G 中的某条边 vw 连接其中的两棵树，即 v 和 w 分属 F 中两个不同连通分量，因此，当算法在处理边 vw 时并未将其加入 F 中，这意味着若将 vw 加入当时产生的树 F' 将构成环路，由前面的引理可知 v 和 w 属于 F' 的同一个连通分量，这与 v 和 w 分属 F 中两个不同连通分量矛盾。由此可知 G 中一个连通分量仅对应 F 中一棵树。

Theorem 4.9. 设 $G = (V, E, W)$ 是带权无向图，$F \subseteq E$，若 F 属于 G 的某个最小生成树集，e 是 $E - F$ 中权值最小的边，并且 $F \cup \{e\}$ 不包含环路，则 $F \cup \{e\}$ 也属于 G 的某个最小生成树集。

Proof. 用反证法，假设 $F \cup \{e\}$ 不属于 G 的最小生成树集，则在 G 的最小生成树集中必然不包含边 e，这表示若将 e 加入 G 的某一最小生成树集 F'，必构成环路，并且该环路上必存在一条边权值大于等于 e 的权值，否则环路上的其他边在考察边 e 之前就已经加入到 F 中，这样 $F \cup \{e\}$ 中就会产生环路，现将这条边换成 e，产生的仍是 G 的生成树集，但其权值小于等于 F'，这与假设矛盾。 □

算法实现细节

- 最小权值边的选取：最小优先队列
- 环路判断：当且仅当 v 和 w 属于 F 的同一个连通分量，vw 加入 F 将构成环路 ⇒ 动态等价关系、合并-查找程序

Algorithm `KruskalMST`(G, n)

```
1  根据 G 的所有边及其权值构造最小优先队列 pq;
2  构造动态等价类集合 sets，G 的每个顶点初始化为一个等价类;
3  F ← ∅;
4  while not IsEmpty(pq) do
5      vw ← GetMin(pq); DeleteMin(pq);
6      vSet ← Find(sets, v); wSet ← Find(sets, w);
7      if vSet ≠ wSet then
8          F ← F ∪ {vw}; Union(sets, vSet, wSet);
9      end
10 end
11 return F;
```

算法分析

- 边优先队列构造：$\Theta(m)$
- 从优先队列中取出和删除所有边：$\Theta(m \log m)$

  - 实际上这个复杂度可以降到 $\Theta(n \log m)$，因为 F 中最多只能有 $n-1$ 条边

- 动态等价类（连通分量）的维护：`Find` 操作最多执行 $2m$ 次，`Union` 最多执行 $n-1$ 次，这部分时间开销至多为 $O((m+n)\lg^*(n))$ (使用 `WUnion` 和 `CFind`)
- 通常情况下 $m \geq n$，因此在最坏情况下，`KruskalMST` 时间复杂度为 $\Theta(m \log m)$,若所有边事先已按权值排好序,则复杂度可降到 $O(m \lg^*(n))$，接近线性
- 与 Prim 算法的比较：
  - Prim 算法：$\Theta(n^2)$，适合处理稠密图（边较多）
  - Kruskal 算法：$\Theta(m \log m)$，适合处理稀疏图（边较少）

5 单源最短路径问题

5.1 基本问题

问题描述

- 问题的由来：
  - 地图上两地点间的最佳行车路径
  - 网络中两台计算机间的最佳路由
- 问题的抽象：
  - 在带权图中查找两个给定顶点间权值和最小的路径（最短路径）
  - 在最坏情况下，寻找两个给定顶点间的最短路径不会比寻找从一个给定顶点到其可达的所有其他顶点间的最短路径更容易
  - 后者被称为**单源最短路径问题**
- 解决思路：问题分割（分治法?）

  Lemma 5.1 (最短路径性质 (shortest path property))**.** 在带权图 G 中，顶点 x 到顶点 z 的最短路径由 x 到 y 的路径 P 和 y 到 z 的路径 Q 组成，则 P 是 x 到 y 的一条最短路径，Q 是 y 到 z 的一条最短路径。注意：反之不成立！

问题描述 (cont.)

Definition 5.2. 设 P 是带权图 $G=(V,E,W)$ 中由 k 条边 $xv_1, v_1v_2, \cdots, v_{k-1}y$ 构成的一条非空路径，则路径 P 的**权值** (weight) 等于构成路径的 k 条边的权值之和，记为 $W(P)$；若 $x=y$ 则称 x 到 y 的路径为空路径，其权值为 0。若从 x 到 y 的路径中没有权值小于 $W(P)$ 的路径，则称 P 为**最短路径** (shortest path) 或**最小权值路径** (minimum-weight path)。

Definition 5.3 (单源最短路径问题). **单源最短路径问题** (single-source shortest path problem) 即对给定带权图 $G=(V,E,W)$ 和图中的某个顶点 s，找到从 s 到图中每个顶点 v 的最短路径。

- 用深度优先搜索可否解决？

5.2 Dijkstra 算法

基本思路

- 采用与 Prim 算法类似的贪心策略
- 但每次在 fringe 顶点中选择的是离出发顶点 s 最近的顶点，而不是像 Prim 算法那样离生成树最近的顶点
- Dijkstra **算法** (Edsger W. Dijkstra, 1959)

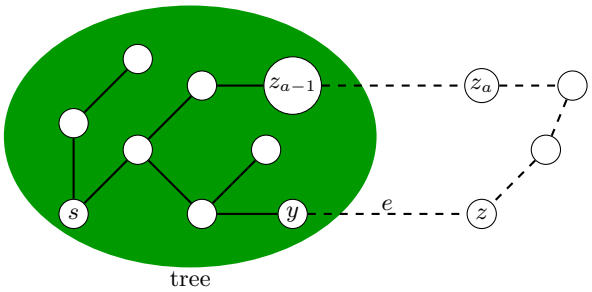
Algorithm `DijkstraSSSP`(G,n)

```
1  初始化所有顶点为 unseen;
2  选择任一顶点 s 开始构造生成树，将其类型改为 tree;
3  d(s,s) ← 0;
4  将与 s 相连的所有顶点类型改为 fringe;
5  while 还有类型为 fringe 的顶点 do
6      找到 fringe 顶点 v 及其相邻 tree 顶点 t 使 d(s,t)+W(tv) 最小;
7      将 tv 加入生成树，并将 v 的类型修改为 tree;
8      d(s,v) ← d(s,t)+W(tv);
9      将所有与 v 相连的类型为 unseen 的顶点类型改为 fringe;
10 end
```

Edsger W. Dijkstra：1930-2002，荷兰著名计算机科学家，其一身的贡献主要集中在程序设计和分布式计算理论领域，1972 年由于其在程序设计语言方面的杰出的基础性的贡献获得图灵奖，ACM Symposium on Principles of Distributed Computing 在他去世后将每年的 Influential Paper Award 更名为 Dijkstra Prize。另外，他还是结构化程序设计理论的创立者和倡导者，最早反对程序设计语言中 GOTO 语句的使用。

算法的正确性保证

Theorem 5.4. 设 $G=(V,E,W)$ 是一个权值非负的带权图，$V'\subseteq V, s\in V'$，若对每一个 $y\in V'$，$d(s,y)$ 是从 s 到 y 的最短距离，若 yz 是 $y\in V'$，$z\in V-V'$ 的所有边中使得 $d(s,y)+W(yz)$ 最小的边，则 s 到 y 的路径加上边 yz 构成 s 到 z 的最短路径。



算法的正确性保证 (cont.)

Proof. 设 $P = s, x_1, \cdots, x_r, y, z$，其中，$s, x_1, \cdots, x_r, y$ 是对应于 $d(s, y)$ 的从 s 到 y 的最短路径；假设 $P' = s, z_1, \cdots, z_a, \cdots, z$ 是从 s 到 z 的最短路径，z_a 是路径中第一个不在 V' 中的顶点，则：

$$W(P) = d(s, y) + W(yz) \leq d(s, z_{a-1}) + W(z_{a-1}z_a)$$

而根据前面的引理，$s, z_1, \cdots, z_{a-1}$ 是从 s 到 z_{a-1} 的最短路径，因此其权值为 $d(s, z_{a-1})$，又因为 $s, z_1, \cdots, z_{a-1}, z_a$ 是路径 P' 的一部分，且边的权值非负，所以：

$$d(s, z_{a-1}) + W(z_{a-1}z_a) \leq W(P')$$

所以，$W(P) \leq W(P')$。 □

算法实现

- 具体实现与 Prim 算法类似
- 复杂度分析完全一样

Algorithm `DijkstraSSSP`(G, n)

```
1 初始化所有顶点为 unseen，优先队列 pq 为空;
2 选择任一顶点 s 设置其候选边为 (−1, s, 0);
3 Insert(pq, s, 0);
4 while pq 非空 do
5     v ← GetMin(pq);
6     DeleteMin(pq);
7     将 v 的候选边加入生成树，v 的类型修改为 tree;
8     UpdateFringe(pq, G, v);
9 end
```

`UpdateFringe`

Procedure `UpdateFringe`(pq, G, v)

```
1  vDist ← GetPriority(pq, v);
2  foreach 与 v 相连的顶点 w do
3      wDist ← vDist + W(v, w);
4      if w 类型为 unseen then
5          将 w 的候选边设置为 (v, w, wDist);
6          w 的类型修改为 fringe;
7          Insert(pq, w, wDist);
8      else if w 的类型为 fringe and wDist < GetPriority(pq, w)
        then
9          将 w 的候选边改为 (v, w, wDist);
10         DecreaseKey(pq, w, wDist);
11     end
12 end
```