离散数学

Discrete Mathematics

极夜酱

目录

Chapter 1 图

1.1 图

1.1.1 图（Graph）

你的微信中有若干好友，而你的好友又有若干好友。许许多多的用户组成了一个多对多的关系网，这个关系网就是数据结构中的图。

再例如使用地图导航功能时，导航会根据你的出发地和目的地规划最佳的地铁换乘路线。许许多多的地铁站组成的交通网络也可以认为是图。

图是一种比树更为复杂的数据结构。树的结点之间是一对多的关系，并且存在父与子的层级划分。而图的顶点之间是多对多关系，并且所有顶点都是平等的，无所谓谁是父子。

在图中，最基本的单元是顶点（vertex），相当于树中的结点。顶点之间的关联关系被称为边（edge）。图中包含一组顶点和一组边，通常用 V 表示顶点集合，用 E 表示边集合。边可以看作是顶点对，即 $(v, w) \in E,\ v, w \in V$。

在有些图中，每一条边并不是完全等同的。例如地铁线路，站与站之间的距离都有可能不同。因此图中会涉及边的权重（weight），涉及到权重的图被称为带权图（weighted graph），也称为网络。

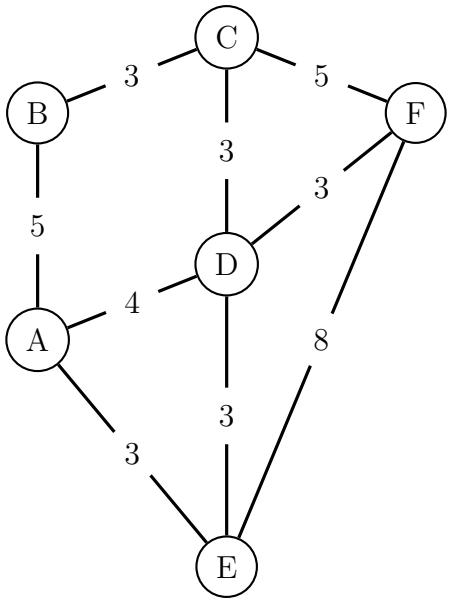


图 1.1: 带权图

还有一种图，顶点之间的关联并不是完全对称的。拿微信举例，你的好友列表里有我，但我的好友列表里未必有你。

这样一来，顶点之间的边就有了方向的区分，这种带有方向的图被称为有向图（directed graph）。有向边可以使用 <v, w> 表示从 v 指向 w 的边。

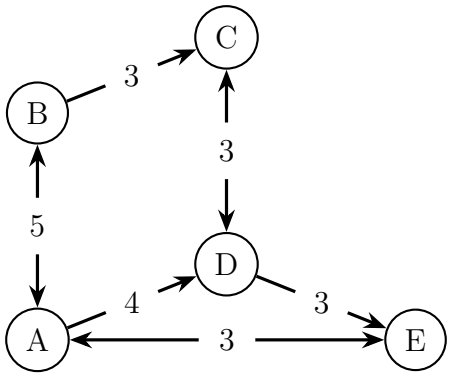


图 1.2: 有向图

相应地，在 QQ 中，只要我把你从好友里删除，你在自己的好友列表里就看不到我了。因此 QQ 的好友关系可以认为是一个没有方向区分的图，这种图被称为无向图（undirected graph）。

1.1.2 图的术语

图还有一些有关路径的术语：

- 度：一个顶点的度是指与该顶点相关联的边的条数。
- 入度：对于有向图，入度为以该顶点为终点的边数。
- 出度：对于有向图，入度为以该顶点为起点的边数。
- 连通：如果从顶点 V 到 W 存在一条路径，则称 V 和 W 是连通的。
- 路径：顶点 V 到 W 的路径是一系列顶点 $\{V, v_1, v_2, \ldots, v_n, W\}$ 的集合，其中任意一对相邻的顶点间都有图中的边。
- 路径长度：路径中边的个数，如果是带权图（网络），则是所有边的权重和。
- 简单路径：顶点 V 到 W 之间的路径中所有顶点都不同。
- 回路：起点等于终点的路径。

握手定理 Handshaking Theorem 假设 $G = (V, E)$ 是无向图，每条边都会给顶点的度之和增加 2，则

$$\sum_{v \in V} deg(v) = 2|E| \tag{1.1}$$

Exercise 一个有 10 个顶点，且每个顶点的度都为 6 的图，有多少条边？

$$2m = 6 \times 10$$

$$m = 30$$

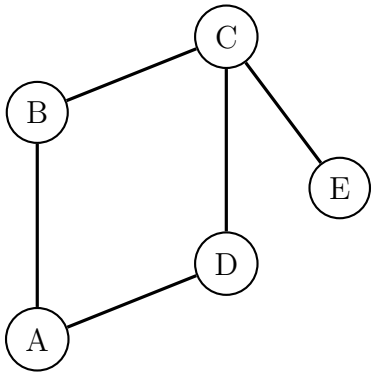
1.2 图的表示

1.2.1 邻接矩阵（Adjacency Matrix）

拥有 n 个顶点的图，它所包含的边的数量最多是 n(n-1) 条，因此，要表达各个顶点之间的关联关系，最清晰易懂的方式是使用邻接矩阵 G[N][N]。

对于无向图来说，如果顶点之间有关联，那么邻接矩阵中对应的值为 1；如果顶点之间没有关联，那么邻接矩阵中对应的值为 0。

$$G[i][j] = \begin{cases} 1 & < v_i, v_j > \text{是 G 中的边} \\ 0 & < v_i, v_j > \text{不是 G 中的边} \end{cases}$$

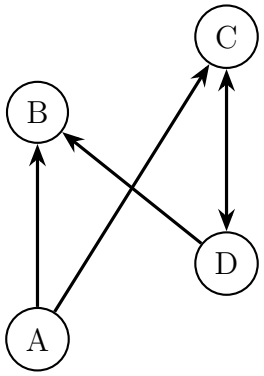


	A	**B**	**C**	**D**	**E**
A	0	1	0	1	0
B	1	0	1	0	0
C	0	1	0	1	1
D	1	0	1	0	0
E	0	0	1	0	0

表 1.1: 无向图邻接矩阵

需要注意的是，邻接矩阵从左上到右下的一条对角线上的元素值必然是 0，因为任何一个顶点与它自身是没有连接的。同时，无向图对应的邻接矩阵是一个对称矩阵，假如 A 和 B 有关联，那么 B 和 A 也必定有关联。

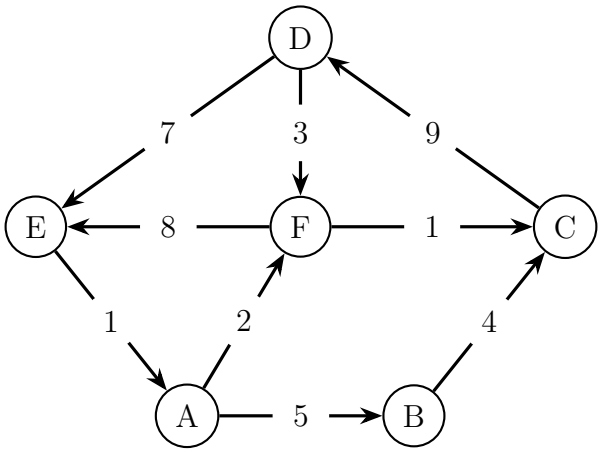
但是对于有向图的邻接矩阵，不一定是一个对称矩阵，假如 A 可以达到 B，从 B 未必能达到 A。



	A	**B**	**C**	**D**
A	0	1	1	0
B	0	0	0	0
C	0	0	0	1
D	0	1	1	0

表 1.2: 有向图邻接矩阵

对于网络，只要把邻接矩阵对应位置的值定义为边 $< v_i, v_j >$ 的权重即可。



	A	**B**	**C**	**D**	**E**	**F**
A	∞	5	∞	∞	∞	2
B	∞	∞	4	∞	∞	∞
C	∞	∞	∞	9	∞	∞
D	∞	∞	∞	∞	7	3
E	1	∞	∞	∞	∞	∞
F	∞	∞	1	∞	8	∞

表 1.3: 带权图邻接矩阵

对于带权图，如果 v_i 和 v_j 之前没有边应该将权值设为 ∞。

邻接矩阵的优点：

1. 简单、直观。
2. 可以快速查到一个顶点和另一顶点之间的关联关系。
3. 方便计算任一顶点的度，对于有向图，从顶点发出的边数为出度，指向顶点的边数为入度。

邻接矩阵的缺点：

1. 浪费空间，对于稀疏图（点很多而边很少）有大量无效元素。但对于稠密图（特别是完全图）还是很合算的。
2. 浪费时间，统计稀疏图中边的个数，也就是计算邻接矩阵中元素 1 的个数。

1.2.2 邻接表（Adjacency List）

为了解决邻接矩阵占用空间的问题，人们想到了另一种图的表示方法——邻接表。在邻接表中，图的每一个顶点都是一个链表的头结点，其后连接着该顶点能够直接到达的相邻顶点。对于稀疏图而言，邻接表存储方式占用的空间比邻接矩阵要小得多。

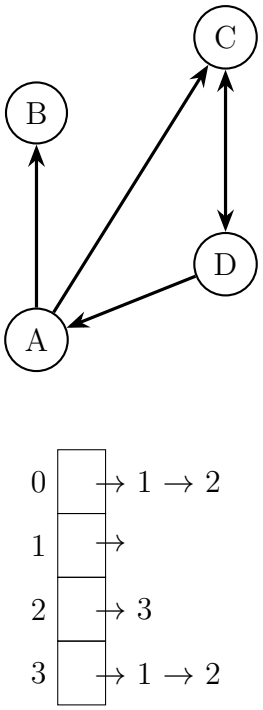


图 1.3: 邻接表

通过遍历邻接表可以查找到所有能够到达的相邻顶点，但是对于逆向查找，即哪些顶点可以达到一个顶点就会很麻烦。

逆邻接表和邻接表是正好相反的，逆邻接表每一个顶点作为链表的头结点，后继结点所存储的是能够直接到达该顶点的相邻顶点。

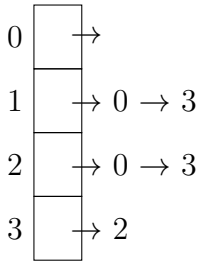


图 1.4: 逆邻接表

可是，一个图要维护正反两个邻接表，也太麻烦了吧?

通过十字链表可以把邻接表和逆邻接表结合在一起。

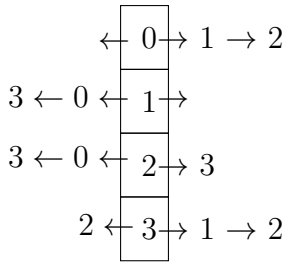


图 1.5: 十字链表

1.3 特殊图

1.3.1 完全图（Complete Graph）

完全图是指每对顶点之间都有一条边的简单图，包含 n 个顶点的完全图记作 K_n。

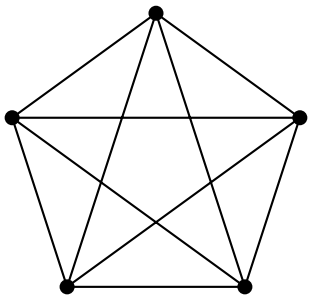


图 1.6: 完全图

1.3.2 圈图（Cycle Graph）

圈图是由 n $(n \geq 3)$ 的顶点，及边 (v_1, v_2)、(v_2, v_3)、(v_{n-1}, v_n)、(v_n, v_1) 组成的简单图。

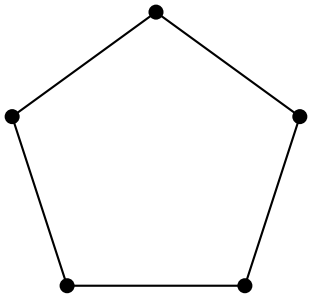


图 1.7: 圈图

1.3.3 n 立方图

n 立方图记作 Q_n，是用顶点表示 2^n 个长度为 n 的二进制串的图。n 立方图的两个顶点相邻，当且仅当它们所表示的二进制串恰有一位不同。

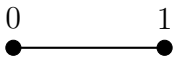


图 1.8: Q_1

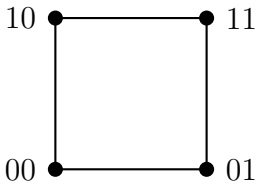


图 1.9: Q_2

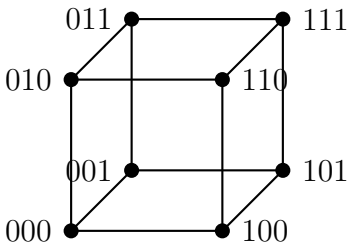


图 1.10: Q_3

1.3.4 二分图（Bipartite Graph）

二分图是指图的顶点可以分成两个不相交的子集，使得每条边都连接一个子集中的顶点与另一个子集中的顶点。

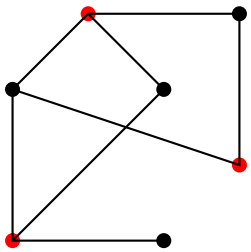


图 1.11: 二分图

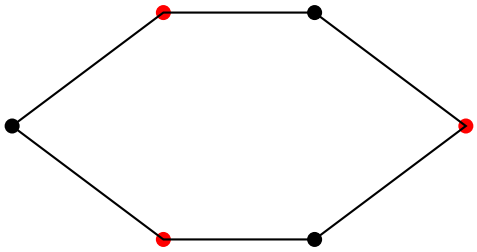


图 1.12: 二分图

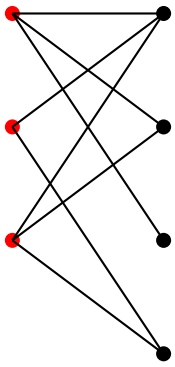


图 1.13: 二分图

如果能够对图中的顶点赋予两种不同的颜色，并且使得没有两个相邻的顶点的颜色是一样的，那么这个图是一个二分图。

如果一个图中存在经过奇数个顶点的环路，那么这个图一定不是二分图。

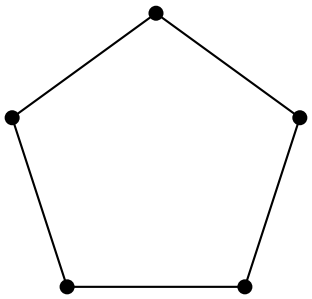


图 1.14: 非二分图

1.4 图的遍历

1.4.1 深度优先搜索（DFS, Depth First Search）

深度优先搜索是一种一头扎到底的遍历方法，选择一条路，尽可能不断地深入，遇到死路就回退，回退过程中如果遇到没探索的支路，就进入该支路继续深入。

例如一个小镇的每个地点都藏有可以实现愿望的光玉，现在要从 0 号出生点出发去收集所有的光玉。

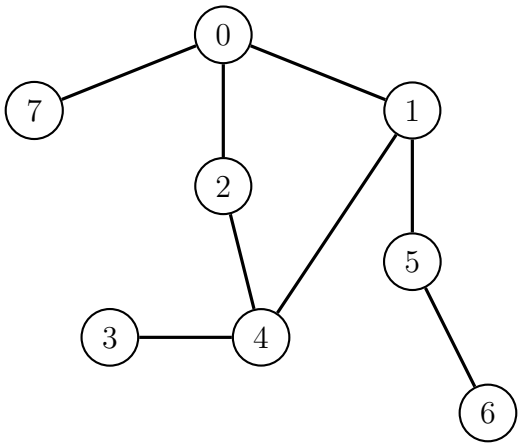


图 1.15: 深度优先搜索

二叉树的先序遍历本质上也可以认为是图的深度优先遍历。要想实现回溯，可以利用栈的先进后出的特性，也可以采用递归的方式，因为递归本身就是基于方法调用栈来实现的。

1.4.2 广度优先搜索（BFS, Breath First Search）

除了深度优先搜索一头扎到底的方法以外，还有一种方法就是首先把从源点相邻的顶点遍历，然后再遍历稍微远一点的顶点，再去遍历更远一点的顶点。

二叉树的层次遍历本质上也可以认为是图的广度优先遍历，需要借助队列来实现重放。

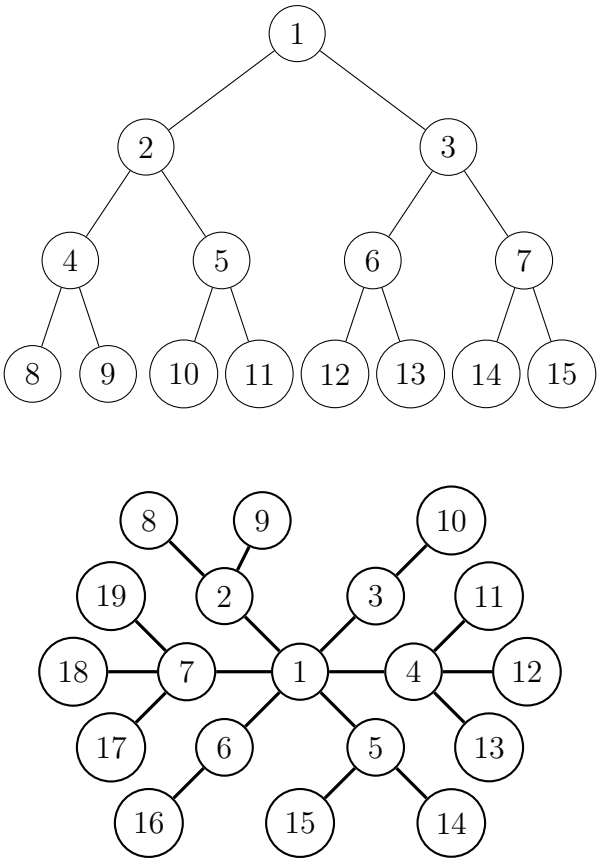

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
8
9
10
19
2
3
11
18
7
1
4
12
17
6
5
13
16
15
14

图 1.16: 广度优先搜索

1.5 连通图

1.5.1 连通图

图还有一些有关路径的术语：

- 连通：如果从顶点 V 到 W 存在一条路径，则称 V 和 W 是连通的。
- 路径：顶点 V 到 W 的路径是一系列顶点 $\{V, v_1, v_2, \ldots, v_n, W\}$ 的集合，其中任意一对相邻的顶点间都有图中的边。
- 路径长度：路径中边的个数，如果是带权图（网络），则是所有边的权重和。
- 简单路径：顶点 V 到 W 之间的路径中所有顶点都不同。
- 回路：起点等于终点的路径。

如果图中任意两顶点均连通，那么称这个图是一个连通图。

一个图的连通分量指的是图的极大连通子图，极大连通子图需要满足两点要求：

1. 顶点数到达极大，即再加一个顶点就不连通了。
2. 边数达到极大，即包含子图中所有顶点相连的所有边。

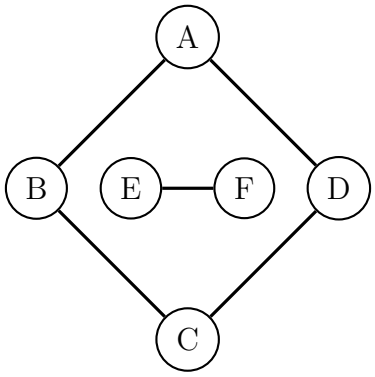


图 1.17: 图 G

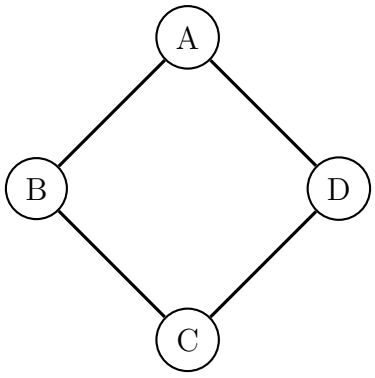


图 1.18: 是图 G 的极大连通子图

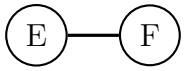


图 1.19: 是图 G 的极大连通子图

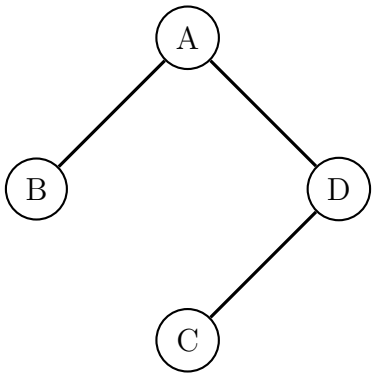


图 1.20: 不是图 G 的极大连通子图

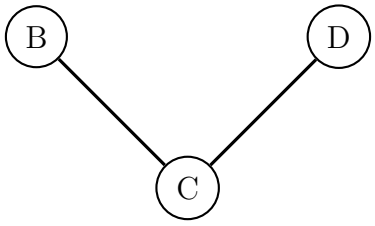


图 1.21: 不是图 G 的极大连通子图

对于有向图而言，如果有向图中任意一对顶点 V 和 W 之间存在双向路径，既可以从 V 走到 W，也可以从 W 走到 V，但这两条路径不一定是同一条，则称该图为强连通图。

如果有向图不是强连通图，但将所有的有向边替换为无向边之后可以变为连通图，则称该图为弱连通图。

有向图的极大强连通子图称为强连通分量。

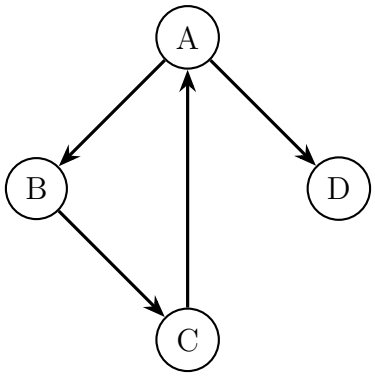


图 1.22: 有向图 G

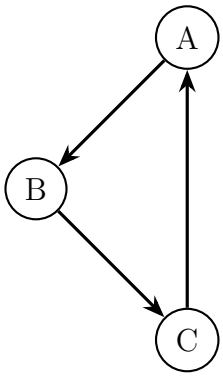


图 1.23: 是有向图 G 的强连通分量

图 1.24: 是有向图 G 的强连通分量

1.6 最短路径

1.6.1 最短路径（Shortest Path）

在现实中很多需要都运用到了最短路径的算法，例如从一个地铁站到另一个地铁站的最快换乘路线等。地铁线路图中，地铁站可以看作是图的顶点，站与站之间的线路可以看作是边，权重可以是距离、时间、费用等。

图 1.25: 上海地铁线路图

在网络中，求两个不同顶点之间的所有路径中，边的权值之和最小的那一条路径，这条路径就是两点之间的最短路径。其中最短路径的第一个顶点称为源点（source），最后一个顶点为终点（destination）。

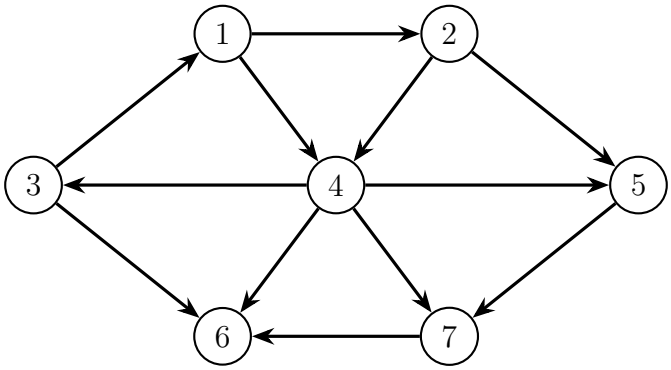
图的最短路径问题分为 2 种类型：

1. 单源最短路径：从某固定源点出发，求到所有其它顶点的最短路径。
2. 多源最短路径：求任意两顶点间的最短路径。

1.6.2 无权图的单源最短路径算法(SSSP, Single-Source Shortest Path)

无权图的单源最短路径算法可以按照递增（非递减）的顺序找出到各个顶点的最短路，算法类似广度优先遍历。

例如在一个无权图中，以顶点 3 作为源点，离源点距离为 1 的顶点有 1 和 6，距离为 2 的顶点有 2 和 4，距离为 3 的顶点有 5 和 7。



无权图的单元最短路径算法中，dist[v] 存储从源点 S 到 v 的最短路径，初始化源点 dist[S] 的距离为 0，path[v] 表示达到顶点路径 v 上一个经过的顶点。

顶点	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dist	1	2	0	2	3	1	3
path	3	1	-1	1	2	3	4

表 1.4: 最短路径表

1.6.3 有权图的单源最短路径算法

有权图的最短路径不一定是经过顶点树最少的路。如果图中存在负值圈(negative-cost cycle）的话会导致算法失效，因为沿着回路走无穷多次，花销是负无穷。

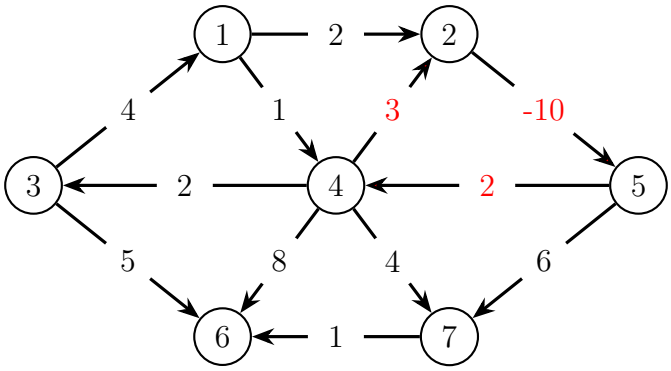


图 1.26: 负值圈

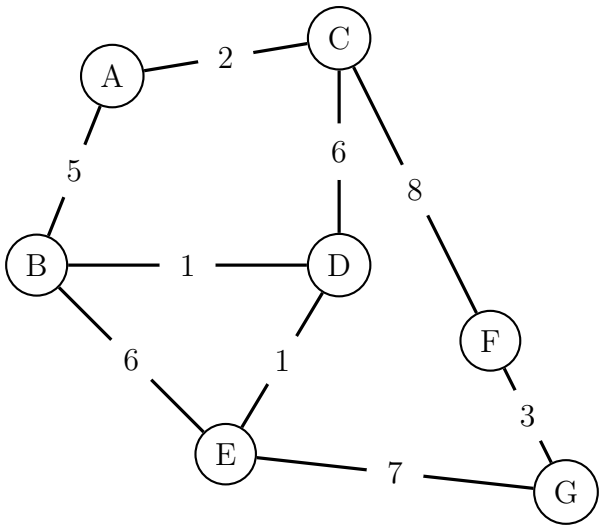
带权图的单源最短路径可以通过迪杰斯特拉（Dijkstra）算法解决。

特斯拉？什么鬼？

迪杰斯特拉算法的本质是不断刷新起点与其他各个顶点之间的距离表。Dijkstra 算法采用了贪心的思想，每次都未收录的顶点中选取 dist 值最小的收录。每当收录一个顶点时，可能会影响另外一个顶点的 dist 值。

$$dist[w] = min\{dist[w], dist[v] + weight_{<v,w>}\}$$

例如计算从源点 A 到其它各顶点的最短路径。



第 1 步：创建距离表。其中表中 key 是顶点名称，value 是源点 A 到对应顶点的已知最短距离。一开始并不知道最短路径是多少，因此 value 都为 ∞。

B	C	D	E	F	G
∞	∞	∞	∞	∞	∞

第 2 步：找到源点 A 的邻接点 B 和 C，从 A 到 B 的距离是 5，从 A 到 C 的距离是 2。

B	C	D	E	F	G
5	2	∞	∞	∞	∞

第 3 步：从距离表中找到从 A 出发距离最短的顶点，也就是顶点 C。找到顶点 C 的邻接点 D 和 F（A 已经遍历过不需要考虑）。从 C 到 D 的距离是 6，所以从 A 到 D 的距离是 $2 + 6 = 8$；从 C 到 F 的距离是 8，所以从 A 到 F 的距离是 $2 + 8 = 10$。

B	C	D	E	F	G
5	2	8	∞	10	∞

第 4 步：从距离表中找到从 A 出发距离最短的顶点（C 已经遍历过不需要考虑），也就是顶点 B。找到顶点 B 的邻接点 D 和 E（A 已经遍历过不需要考虑）。从 B 到 D 的距离是 1，所以从 A 到 D 的距离是 $5 + 1 = 6$，小于距离表中的 8；从 B 到 E 的距离是 6，所以从 A 到 E 的距离是 $5 + 6 = 11$。

B	C	D	E	F	G
5	2	6	11	10	∞

第 5 步：从距离表中找到从 A 出发距离最短的顶点（B 和 C 不用考虑），也就是顶点 D。找到顶点 D 的邻接点 E 和 F。从 D 到 E 的距离是 1，所以从 A 到 E 的距离是 $6 + 1 = 7$，小于距离表中的 11；从 D 到 F 的距离是 2，所以从 A 到 F 的距离是 $6 + 2 = 8$，小于距离表中的 10。

B	C	D	E	F	G
5	2	6	7	8	∞

第 6 步：从距离表中找到从 A 出发距离最短的顶点，也就是顶点 E。找到顶点 E 的邻接点 G。从 E 到 G 的距离是 7，所以从 A 到 G 的距离是 $7 + 7 = 14$。

B	C	D	E	F	G
5	2	6	7	8	14

第 7 步：从距离表中找到从 A 出发距离最短的顶点，也就是顶点 F。找到顶点 F 的邻接点 G。从 F 到 G 的距离是 3，所以从 A 到 G 的距离是 $8+3=11$，小于距离表中的 14。

B	C	D	E	F	G
5	2	6	7	8	11

最终，距离表中存储的是从源点 A 到所有顶点的最短距离。

1.6.4 多源最短路径算法

如何能够求出一个带权图中所有顶点两两之间的最短距离呢?

对了！刚刚学过了 Dijkstra 算法，可以对每个顶点都使用一次 Dijkstra 算法，这样就求出了所有顶点之间的最短距离。

这个思路确实可以实现，但是 Dijkstra 算法的代码逻辑比较复杂，有没有更简单的方法呢?

弗洛伊德（Floyd-Warshall）算法是专门用于寻找带权图中多源点之间的最短路径算法。Floyd 算法的思想是，若想缩短两点间的距离，仅有一种方式，那就是通过第三顶点绕行。

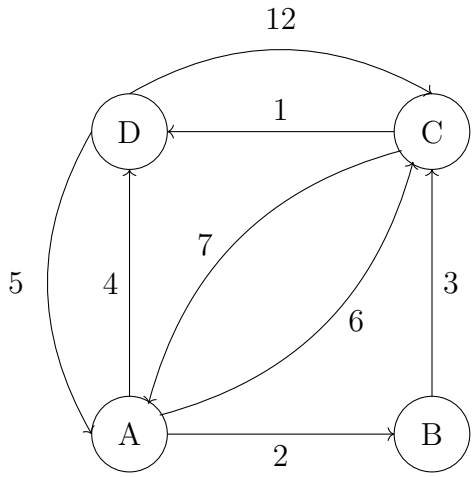
假设 $D^k[i][j]$ 为路径 $\{i \to \{l \le k\} \to j\}$ 的最小长度。当 D^{k-1} 已经完成，递推到 D^k 时:

1. 如果 $k \notin$ 最短路径$\{i \to \{l \le k\} \to j\}$，则 $D^k = D^{k-1}$。
2. 如果 $k \in$ 最短路径$\{i \to \{l \le k\} \to j\}$，该路径必定由两段最短路径组成，则 $D^k[i][j] = D^{k-1}[i][k] + D^{k-1}[k][j]$。

例如，小哼准备去一些城市旅游，有些城市之间有公路，有些城市之间则没有。为了节省经费以及方便计划旅程，小哼希望在出发之前直到任意两个城市之间的最短路程。

如果要让任意两点之间的路程变短，只能引入第三个点，并通过这个顶点中转才有可能缩短原来的路程。这个中转的顶点甚至有时候不只通过一个顶点，而是经过两个或更多点中转会更短。

当任意两点之间不允许经过第三个点中转时，这些城市之间的最短路径就是邻接矩阵的初始路径。



	A	**B**	**C**	**D**
A	0	2	6	4
B	∞	0	3	∞
C	7	∞	0	1
D	5	∞	12	0

在只允许经过 1 号顶点中转的情况下，任意两点之间的最短路程更新为：

	A	**B**	**C**	**D**
A	0	2	6	4
B	∞	0	3	∞
C	7	9	0	1
D	5	7	11	0

在只允许经过 1 号和 2 号顶点中转的情况下，任意两点之间的最短路程更新为：

	A	B	C	D
A	0	2	5	4
B	∞	0	3	∞
C	7	9	0	1
D	5	7	10	0

在只允许经过 1 号、2 号和 3 号顶点中转的情况下，任意两点之间的最短路程更新为：

	A	B	C	D
A	0	2	5	4
B	10	0	3	4
C	7	9	0	1
D	5	7	10	0

最后允许通过所有顶点作为中转，任意两点之间的最短路程更新为：

	A	B	C	D
A	0	2	5	4
B	9	0	3	4
C	6	8	0	1
D	5	7	10	0

1.7 最小生成树

1.7.1 最小生成树（MST, Mininum Spanning Tree）

所谓最小生成树，就是一个图的极小连通子图，它包含原图的所有顶点，并且所有边的权值之和尽可能小。

最小生成树需要满足 3 个条件：

1. 是一棵树：树不能有回路，并且 $|V|$ 个顶点一定有 $|V|-1$ 条边。
2. 是生成树：包含原图的全部顶点，树的 $|V|-1$ 条边都必须在图里，并且如果向生成树中任意加一条边都一定构成回路。
3. 边的权重和最小。

如果最小生成树存在，那么图一定连通，反之亦然。

例如一个带权图，蓝色边可以把所有顶点连接起来，又保证边的权值和最小。

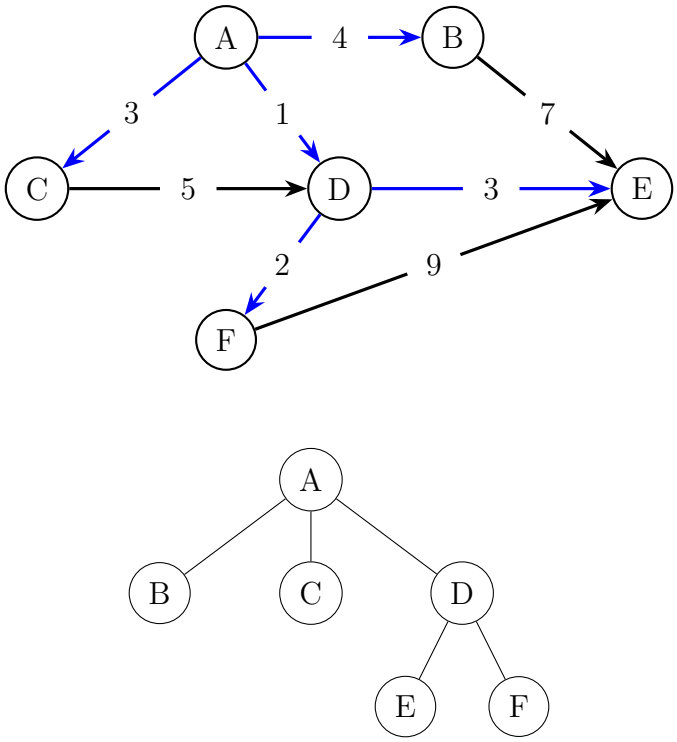


图 1.27: 最小生成树

图的最小生成树不是唯一的，同一个图有可能对应多个最小生成树。

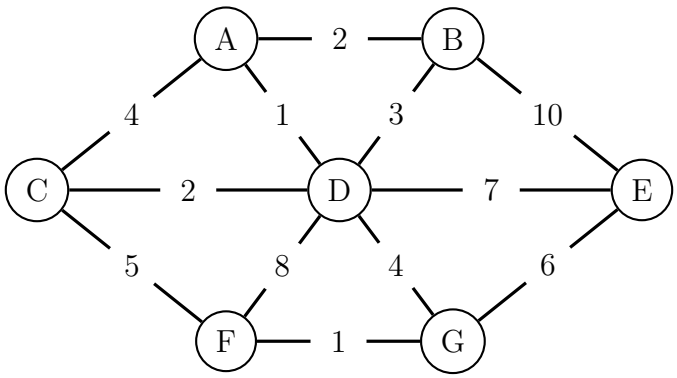
最小生成树在现实中很很多用处。假如要在若干个城市之间铺设铁路，而预算又是有限的，那么就需要寻找成本最低的铺设方式。城市之间的交通网就像一个连通图，其实并不需要在每两个城市之间都直接进行连接，只需要一个最小生成树，保证所有的城市都有铁路可以达到即可。

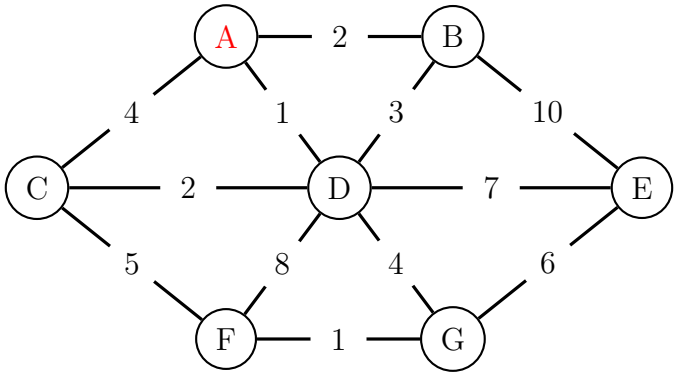
1.7.2 Prim

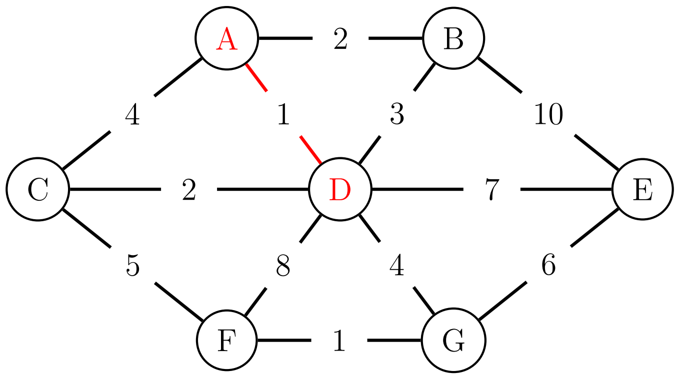
Prim 算法是以图的顶点为基础，从一个初始顶点开始，寻找达到其它顶点权值最小的边，并把该顶点加入到已触达顶点的集合中。当全部顶点都加入到集合时，算法的工作就完成了。Prim 算法的本质是基于贪心算法（greedy algorithm）。

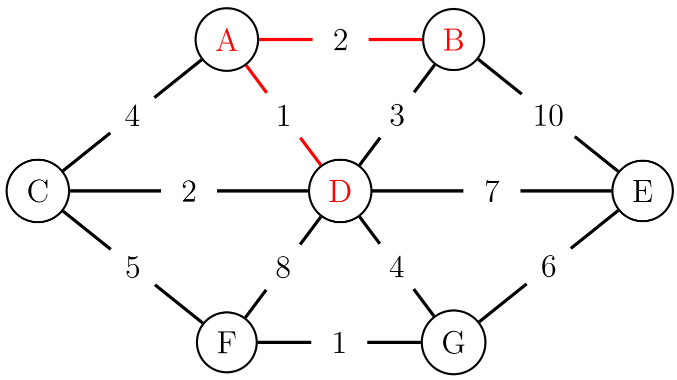
Prim 算法可以理解为让一棵小树长大，每次找能够向外生长的最小边。

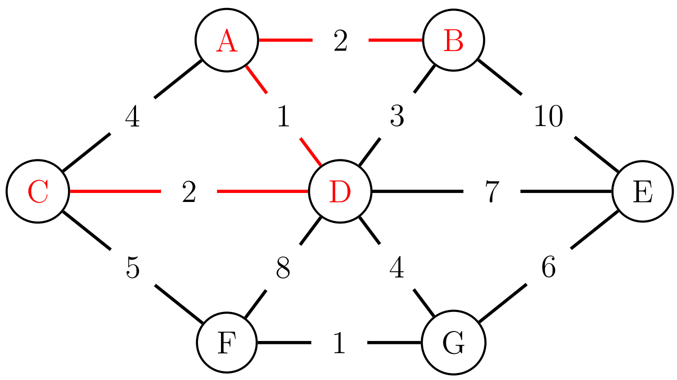
例如使用 Prim 算法获得一个带权图的最小生成树：

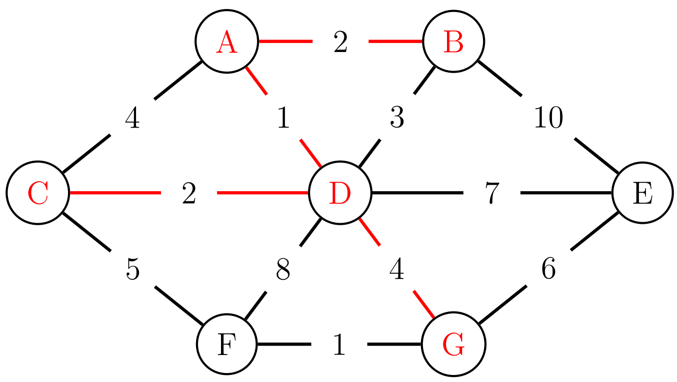


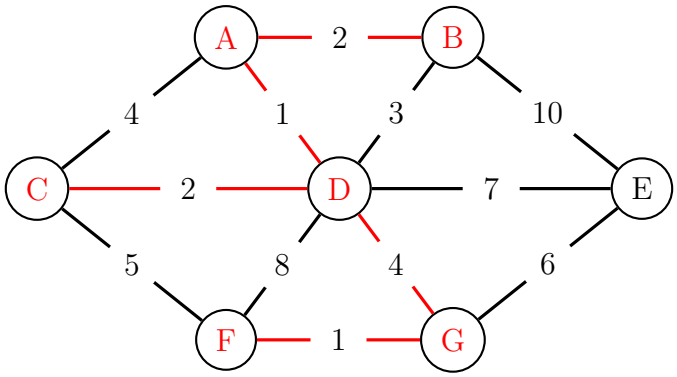


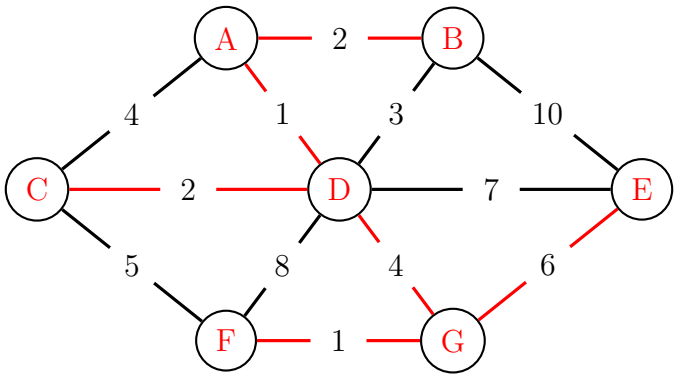

A
2
B
4
1
3
10
C
2
D
7
E
5
8
4
6
F
1
G


A
2
B
4
1
3
10
C
2
D
7
E
5
8
4
6
F
1
G


A
2
B
4
1
3
10
C
2
D
7
E
5
8
4
6
F
1
G


A
2
B
4
1
3
10
C
2
D
7
E
5
8
4
6
F
1
G



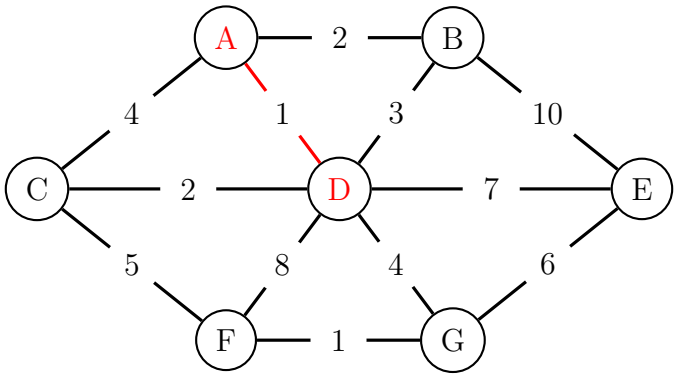


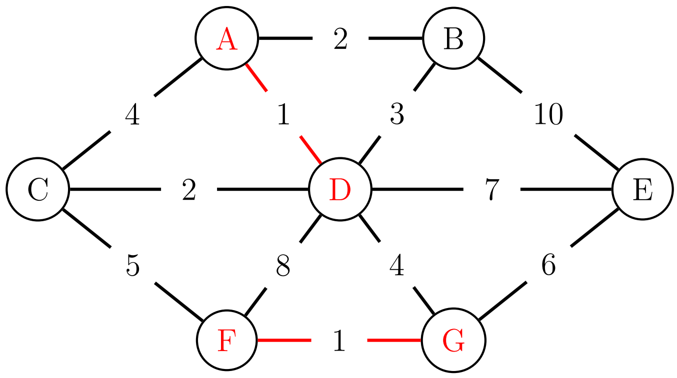
1.7.3 Kruskal

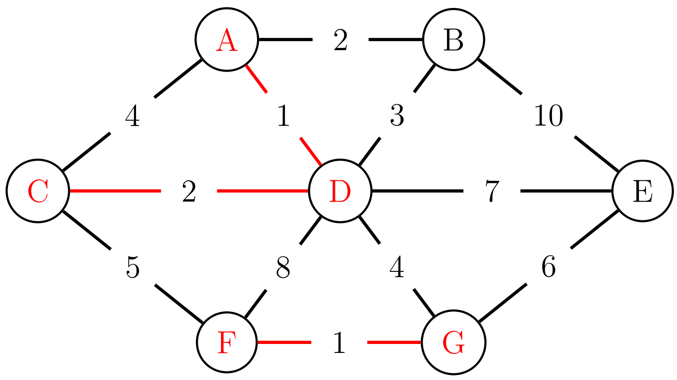
与 Prim 算法不同，Prim 算法是以顶点为关键来获得最小生成树的，而 Kruskal 算法是以边为关键获得最小生成树的。

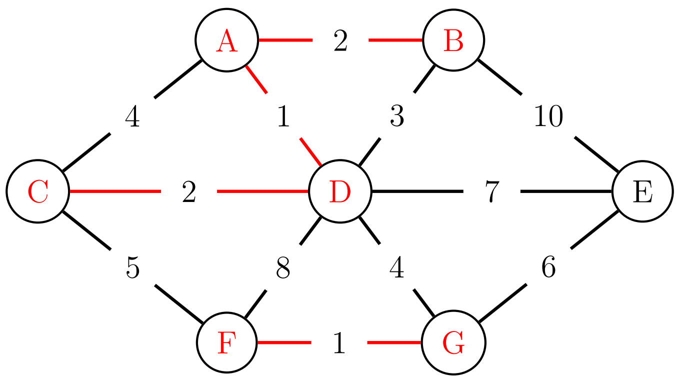
Kruskal 算法可以理解为将森林合并成树，每次在图中找权值最小的边收录。

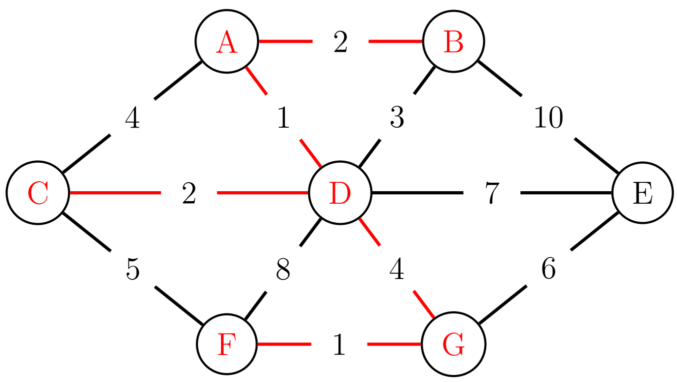
例如使用 Kruskal 算法获得一个带权图的最小生成树：

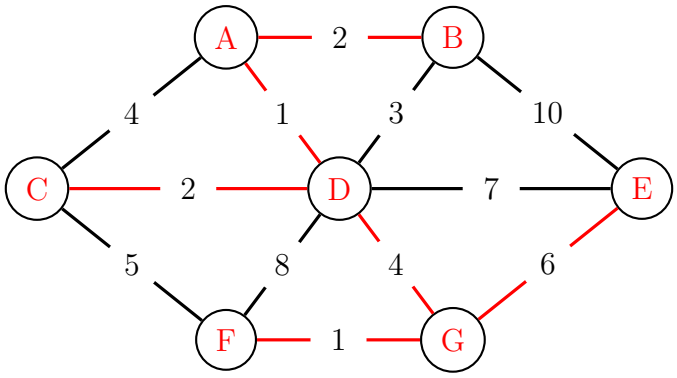



A
B
C
D
E
F
G
2
4
1
3
10
2
7
5
8
4
6
1


A
B
C
D
E
F
G
2
4
1
3
10
2
7
5
8
4
6
1


A
B
C
D
E
F
G
2
4
1
3
10
2
7
5
8
4
6
1


A
B
C
D
E
F
G
2
4
1
3
10
2
7
5
8
4
6
1


A
B
C
D
E
F
G
2
4
1
3
10
2
7
5
8
4
6
1