Chapter 7

图算法

本章是承上启下的一章，我们将介绍一些简单的图算法，这其中有一部分内容将作为后面的回溯法和分支限界的基础，另一部分则是之前我们介绍的动态规划和贪婪策略的应用。

7.1 图的表示

一个图 $G = (V, E)$ 可以由邻接表和邻接矩阵两种方法来表示。其中，邻接表适合表示稀疏图，而邻接矩阵则适合表示稠密图。图还分为有向图和无向图，对于有向图而言，如果存在一条边 $e = (u, v)$，那么在邻接矩阵 Adj 中，$Adj[u][v] = 1$，在邻接表中，结点 v 在结点 u 的链表中。而对于无向图而言，如果存在一条无向边 $e = \{u, v\}$，那么我们将其表示为两条有向边 $e_1 = (u, v), e_2 = (v, u)$。

7.2 深度优先搜索

首先我们要介绍的是图的深度优先搜索。正如其名称一样，图的深度优先搜索在遍历图的过程中尽可能的“深入”。如果我们在遍历到某个结点时，该结点还有与之相邻的结点没有被遍历，那么我们就遍历它，直到遍历不下去为止，此时我们回溯到上一个结点，然后继续找上一个结点有没有还没被遍历过的相邻结点。深度优先搜索将重复该过程直到没有结点没有被遍历过。

在深度优先搜索的过程中，我们对结点的颜色进行涂色。初始状态下，图中的所有结点都是白色的，当这个结点在被遍历到的时候我们将它涂成灰色，当这个结点的所有邻接结点被扫描结束后，我们将它涂成黑色。此外，我们还将结点赋上两个时间戳，其中，$u.d$ 为发现结点 u 的时刻，而 $u.f$ 为完成对 u 的处理的时刻。最后，我们在深度优先搜索的过程中，遍历每个结点 u 时我们都记录它的前驱结点 $u.\pi$，在深度优先搜索结束的时候，我们将每个结点的前驱结点与它连接在一起构成一条有向边，这条有向边由前驱结点发出，指向后继结点，便可以构成一个深度优先森林。它由若干棵深度优先树

组成。该过程如算法 DFS 所示，其中 *time* 是一个全局变量，为时间戳。

DFS(G)

```
1: for each vertex u ∈ G.V do
2:     u.color = WHITE
3:     u.π = NIL
4: time = 0
5: for each vertex v ∈ G.V do
6:     if u.color == WHITE then
7:         DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1: time = time + 1
2: u.d = time
3: u.color = GRAY
4: for each v ∈ G.Adj[u] do
5:     if v.color == WHITE then
6:         v.π = u
7:         DFS-VISIT(G, v)
8: u.color = BLACK
9: time = time + 1
10: u.f = time
```

该算法的时间复杂度是 $\Theta(|V| + |E|)$，因为每个结点和每条边都被访问一次。

如果在深度优先森林的基础上，我们将图中原有的其它边补充进来，并按探索的顺序表明方向，那么图当中的所有边可以分为以下这么几类:

(1) 树边：顾名思义就是在深度优先树当中的边。

(2) 后向边：是由深度优先树中的子孙结点指向祖先结点的边。

(3) 前向边：是由深度优先树中除了树边之外的由祖先结点指向子孙结点的边。

(4) 横向边：其它的所有边。

在深度优先搜索进行的过程中，就已经有足够的信息来判断边的种类了：在第一次探索边 (u, v) 时，如果结点 v 的颜色是白色，则说明这条边是一个树边，如果结点 v 是灰色，则说明这条边是一个后向边，如果结点 v 是黑色，则说明这条边要么是一条前向边，要么是一条横向边，此时如果 $u.d < v.d$，那么是前向边，否则是横向边。

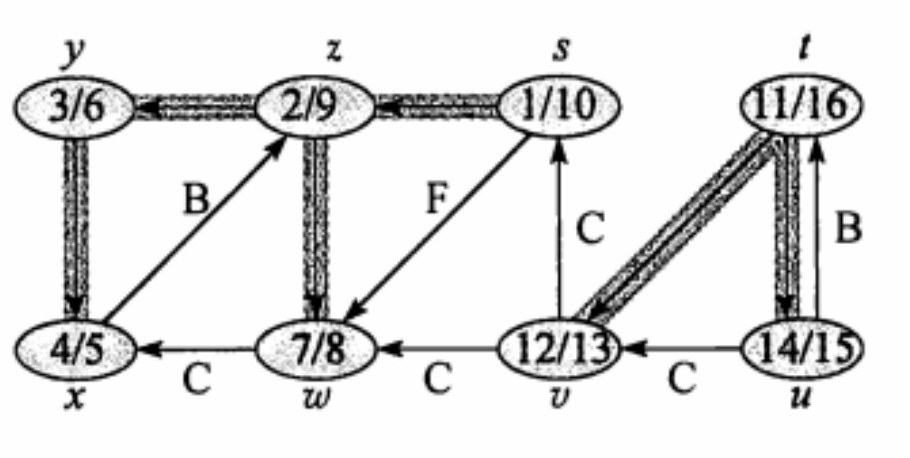

Figure 7.1: 深度优先搜索的示例

接下来我们证明: 在无向图的深度优先搜索树中，不存在前向边和横向边。我们设 (u, v) 为任意一条边，不失一般性的我们假设 $u.d < v.d$，否则由于 (u, v) 是无向边，直接将边的观察方向反转即可，此时结点 v 在结点 u 的邻接链表中，算法在处理完 u 之前一定会完成对结点 v 的处理，如果算法在第一次探索边 (u, v) 时则一定分为两种情况，一种情况是由结点 u 出发探索结点 v，此时 v 结点为白色，那么这条边就是一条树边，否则就是从结点 v 出发探索结点 u，此时 u 结点为灰色，那么 (v, u) 将出现在深度优先树中，为一条后向边。图7.1给出了一幅图做深度优先搜索的示例，结点中的数字是时间戳，B 代表后向边，F 代表前向边，C 代表横向边，加粗的是树边。

7.3 广度优先搜索

接下来我们介绍广度优先搜索，广度优先搜索通常用于解决图上边长为 1 的最短路径问题。为了应用广度优先搜索，我们通常要知道源结点是哪一个，然后我们以源结点为广度优先搜索的起点运行该算法，算法结束的时候，每个结点到源结点的最短路径将被求出。

与深度优先搜索不同，广度优先搜索不是尽可能的深入，而是只有在结点 u 的所有邻接结点都被遍历完时，才结束对该结点的访问。与深度优先搜索一样的是，我们在访问结点的时候，也给结点涂色，初始情况下，所有结点都是白色；当结点正在被访问的时候，将它涂成灰色；而结束访问该结点时，我们将该结点涂成黑色。同时我们记录每个结点 u 的前驱结点 $u.\pi$，当广度优先搜索结束时，从目标结点开始递归的访问它的前驱结点，直到到达源结点为止，这条路径的反向路径则是从源结点到该结点的最短路径。算法 BFS 描述了广度优先搜索，它利用队列来实现，除了接受图的邻接链表或者邻接矩阵作为参数以外，还接受一个源结点 s 作为参数。

BFS(G, s)

```
1: for each v ∈ G.V do
2:     v.color = WHITE
3:     v.π = NIL
4:     v.d = ∞
5: Q = {s}
6: s.color = GRAY
7: s.d = 0
8: while Q ≠ ∅ do
9:     u = DEQUEUE(Q)
10:    for each v ∈ G.Adj[u] do
11:        if v.color == WHITE then
12:            ENQUEUE(Q, v)
13:            v.color = GRAY
14:            v.d = u.d + 1
15:            v.π = u
16:    u.color = BLACK
```

该算法的时间复杂度是 $O(|V|+|E|)$，因为每个结点和边都至多访问一次。其中，每个结点至多被访问一次是因为每个结点只有在进入队列的时候被标记为灰色，而只有结点是白色的时候才会进入队列。每条边至多被访问一次的原因是每个结点被访问的时候，会遍历它的所有边，而每个结点不会被重复访问，所以每条边也不会被重复访问。

接下来我们来讨论广度优先搜索的两个性质。第一个性质是，广度优先搜索进行的过程中，如果结点 u 比结点 v 先进入队列，那么有 $u.d \le v.d$，于是如果我们把结点按进入队列的顺序排列，那么它们的 d 值应该是单调不下降的。

第二个性质就是最短路径的性质了：如果我们记 $\delta(s,u)$ 为从源结点 s 到结点 u 的最短路径距离，那么我们有 $\delta(s,u) = u.d$。如果 u 是从 s 出发不可达的，那么我们有 $u.d = \delta(s,u) = \infty$，此时该性质成立。我们主要讨论从 s 出发，对于每个可达的结点 u，有 $\delta(s,u) = u.d$。我们将使用数学归纳法来证明这个性质，第一步，我们定义谓词 $P(n)$:

- $P(n)$: 若从结点 s 出发至结点 u 的最短路径长度为 n，即 $\delta(s,u) = n$，则完成广度优先搜索后，$u.d = n$。

第二步，证明基本情况 $P(0)$，若从结点 s 出发至结点 u 的最短路径长度为 0，那么说明 $u = s$，于是此时有 $u.d = 0$，因此基本情况成立。第三步，证明一般情况 $P(n) \Rightarrow P(n+1)$，这里我们再用一个反证法来证明这个情况。首先，我们注意到对于每个结点 v，都有 $\delta(s,v) \le v.d$，那么我们接下来就证明对于任意一个结点 v 满足 $\delta(s,v) = n+1$，$\delta(s,v) < v.d$ 是不存在的。首先假设 u 是从 s 到 v 的最短路径上的前驱结点，那么此时 $\delta(s,u) = n$，由归纳假设我们有 $u.d = \delta(s,u) = n$，同时我们有 $\delta(s,v) = \delta(s,u) + 1$。若 $\delta(s,v) < v.d$，那

么有:

$$v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1$$

即 $v.d > u.d + 1$。下面我们考察当 u 结点在第 9 行被取出的时候，并且在第 10 行遍历到 v 结点的情况，此时 v 结点可能是任何一种颜色。如果此时 v 结点是白色的，那么有 $v.d = u.d + 1$，这与刚才的不等式矛盾。如果此时 v 结点是黑色的，那么 v 结点应该在 u 结点之前被访问，此时 $v.d \le u.d$，这也与刚才的不等式矛盾。最后，如果此时 v 结点是灰色的，那么一定在此之前，存在一个结点 w 在第 9 行被从队列取出的时候，将 v 加入了队列，也就是说 w 比 u 要更早的进入队列且此时 $v.d = w.d + 1$，因为 w 比 u 更早的进入队列，所以 $w.d = v.d - 1 \le u.d$，即 $v.d \le u.d + 1$，这仍然与刚才的不等式矛盾。于是不存在 $\delta(s,v) < v.d$，因此 $\delta(s,v) = v.d$。

7.4 拓扑排序

接下来我们介绍有向无环图的拓扑排序。拓扑排序是有向无环图 G 的一种线性次序，该次序满足的条件是：如果图 G 中存在一条边 (u,v)，那么结点 u 在拓扑排序中排在结点 v 的前面。如果在一个有向图 G 中包含一个环路，那么不可能存在一个拓扑排序。

如图7.2所示，图 (a) 中表示的是一个人穿衣服的先后次序，图中的每个结点是穿着种类，有向边表示的是依赖关系，有向边 (u,v) 表示在穿 v 之前要先穿 u，例如，穿鞋之前要先穿袜子。图 (b) 则是一个可行的穿衣次序，即拓扑排序。

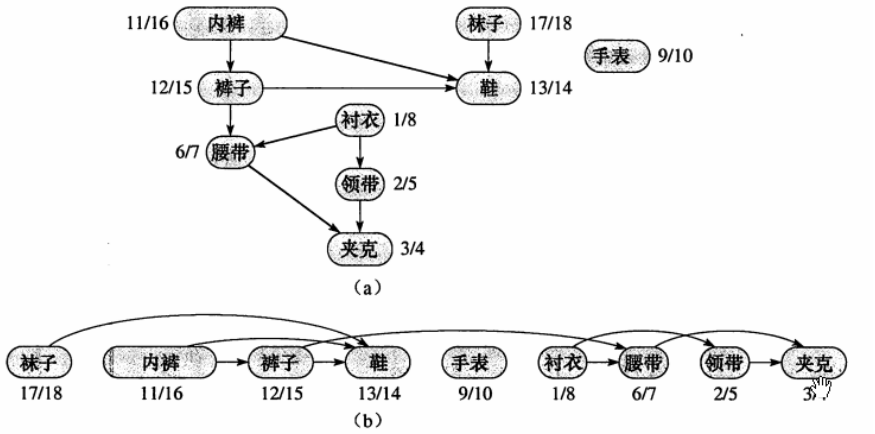

Figure 7.2: 一个拓扑排序的例子

接下来我们证明，一个有向图不存在环当且仅当它不存在后向边。我们先证明 $\Rightarrow$ 方向，如果 G 是一个有向无环图，如果它存在一条后向边 (u,v)，那么在图 G 的深度优先树中，v 是 u 的祖先，也就是说，在深度优先树中存在 $v \rightsquigarrow u$ 的一条路径，加上边 (u,v) 则出现一个环，这与 G 是一个有向无环图矛盾，所以不存在后向边。

然后我们证明 $\Leftarrow$ 方向，即如果一个有向图不存在后向边，则这个有向图不存在环。假设图 G 存在一个环路 c，设结点 v 是环路 c 上第一个被遍历的结点，设 (u,v) 是环路 c 中结点 v 前面的一条边，在时刻 $v.d$，在环路 c 中的结点形成一条从结点 v 到 u 的一条纯白色路径。那么在深度优先树中，结点 u 是结点 v 的后代，即结点 v 是结点 u 的祖先，那么 (u,v) 是一条后向边，这与该图不存在后向边相矛盾。

接下来我们证明，在有向无环图中，如果存在一条边 (u,v)，那么深度优先搜索后，$u.f > v.f$。首先我们刚才已经证明了有向无环图中不存在后向边，因此在深度有限搜索的时候，如果存在一条边 (u,v)，那么我们在遍历到 u 时，v 要么是白色的，要么是黑色的。如果 v 是白色的，那么 v 在深度优先树中是 u 的后代，此时 $u.d < v.d < v.f < u.f$，即 $u.f > v.f$。如果 v 是黑色的，那么 $v.f$ 已经被赋值了，而 $u.f$ 还没有被赋值，那么此时 $u.f > v.f$。

于是，我们可以利用深度优先搜索，首先判断这个图里是否存在后向边，如果存在，则该图里存在一个有向环，此时不存在拓扑排序。否则我们将结点按 f 值逆向排序，就得到了该图的拓扑排序。

7.5 最短路问题

接下来我们考虑图上的最短路问题，所谓最短路问题，就是求解出图 G 中的两个结点 v_i 到 v_j 的最短路径。在这里我们要注意到，最短路径中不存在环。这里我们要分情况讨论，首先如果环是负的，那么这条最短路径是不存在的，因为我们可以在这个负环上无限的走下去，使得总的权值越来越短。此外，也不能存在一个正权环，因为如果我们把这个环去掉，则能够得到一条更短的路径，这与这条路是最短路径相矛盾。最后，如果这个环的总权值为 0，那么我们可以去掉这个环，得到另外一条不包含环的最短路径，我们就研究这条最短路径就行了。因此，最短路径中是不存在环的。

对于最短路问题可以分为单源最短路径问题和任意点对之间的最短路径问题。我们首先证明最短路问题满足最优子结构的性质，证明了这一点，我们便可以使用动态规划策略或者贪婪策略来解决最短路问题了。接下来我们分别讨论单源最短路径中，存在负权边和不存在负权边的两种情况下要怎么求。然后我们讨论任意点对之间的最短路径问题。

7.5.1 最优子结构性质

最短路问题满足最优子结构的性质，即如果一条路径 $s \rightsquigarrow p \rightsquigarrow q$ 是最短路径，那么子路径 $p \rightsquigarrow q$ 也是一条最短路径，否则就有另一条更短的路径从 p 出发走到 q，在这里我们需要分情况讨论这条更短的路径，如果这一条更短的路径中的结点不在子路径 $s \rightsquigarrow p$ 中，那么可以直接将新的路径替换到 $p \rightsquigarrow q$，然后再接上 $s \rightsquigarrow p$ 就得到一条更短的路径，这与原路径是最短路径相矛盾，另一种情况是，如果新的更短的 $p \rightsquigarrow q$ 的路径

中存在一个结点 t，同时该结点 t 在子路径 $s \rightsquigarrow p$ 中，那么就有 $p \rightsquigarrow t \rightsquigarrow q$ 比原来的 p 不经过 t 走到 q 的路径更短。我们记 $W(p \rightsquigarrow t \rightsquigarrow q)$ 为从 p 点出发经过 t 点走到 q 的最短路径长度，即 $W'(p \rightsquigarrow q)$ 为从 p 点出发不经过 t 点走到 q 的最短路径长度，那么有

$$W(p \rightsquigarrow t \rightsquigarrow q) = W(p \rightsquigarrow t) + W(t \rightsquigarrow q) < W'(p \rightsquigarrow q)$$

由于最短路径当中不能存在负环，于是有

$$W(p \rightsquigarrow t) + W(t \rightsquigarrow p) \geq 0$$

于是有

$$\begin{aligned} W(s \rightsquigarrow t \rightsquigarrow p \rightsquigarrow q) &= W(s \rightsquigarrow t) + W(t \rightsquigarrow p) + W'(p \rightsquigarrow q) \\ &> W(s \rightsquigarrow t) + W(t \rightsquigarrow p) + W(p \rightsquigarrow t) + W(t \rightsquigarrow q) \\ &\geq W(s \rightsquigarrow t) + W(t \rightsquigarrow q) \end{aligned}$$

这与原路径是最短路径相矛盾，因此最短路问题满足最优子结构的性质。

我们首先记 $\delta(u, v)$ 为从 u 结点走到 v 结点的最短距离。然后我们定义对一条边 (u, v) 的松弛操作，我们记 $u.d$ 为从源结点 s 走到结点 u 的最短路径的距离的估计值，其中 $w(u, v)$ 表示边 (u, v) 的权值，松弛操作如 RELAX 所示，它的意思是如果从 s 走到 v 的距离大于先沿着从 s 到走到 u 的最短路径走到 u，再沿着 (u, v) 从 u 走到 v 的距离，那么我们就更新它。

RELAX(u, v, w)

1: if $v.d > u.d + w(u, v)$ then
2: $\quad v.d = u.d + w(u, v)$
3: $\quad v.\pi = u$

接下来我们介绍松弛操作和最短路径的几个基本性质（来自算法导论）:

1. 三角不等式性质：对于任何边 $(u, v) \in E$，有 $\delta(s, v) \leq \delta(s, u) + w(u, v)$。
2. 非路径性质：如果从结点 s 到结点 v 之间不存在路径，则有 $v.d = \delta(s, v) = \infty$。
3. 收敛性质: 对于某些结点 $u, v \in V$, 如果 $s \rightsquigarrow u \rightarrow v$ 是图 G 中的一条最短路径, 并且对边 (u, v) 进行松弛前的任意时间有 $u.d = \delta(s, u)$，则在之后的时间有 $v.d = \delta(s, v)$。
4. 路径松弛性质：如果 $p = \langle v_0, v_1, \cdots, v_k \rangle$ 是从源结点 $s = v_0$ 到结点 v_k 的一条最短路径，并且我们对 p 中的边所进行松弛的次序为 (v_0, v_1)，(v_1, v_2)，$\cdots$，(v_{k-1}, v_k)，则 $v_k.d = \delta(s, v_k)$。该性质的成立与任何其它的松弛操作无关，即使这些松弛操作是与对 p 上的边所进行的松弛操作是穿插进行的。
5. 前驱子图性质：对于所有的结点 $v \in V$，一旦 $v.d = \delta(s, v)$，则前驱子图是一棵根结点为 s 的最短路径树。

7.5.2 Dijkstra 算法

这一小节我们将介绍 Dijkstra 算法，用以解决不存在负权边的最短路问题，这是一个应用贪婪策略的算法。Dijkstra 算法的思路是: 初始情况下，除了 $s.d = 0$，对于其余的结点 u 而言，$u.d = \infty$，然后反复地在每次迭代中，选择从源结点能够走到的最近的结点，然后从该结点开始松弛它的邻接结点，此后算法永远不再可能继续选择该结点去再次松弛它的邻接结点了，因此算法在松弛完每个结点后会停止，最终有 $\forall v \in V, \delta(s, v) = v.d$，该算法如 DIJKSTRA 所示。

DIJKSTRA(G, w, s)

1: INITIALIZE-SINGLE-SOURCE(G, s)
2: $S = \varnothing$
3: $Q = G.V$
4: while $Q \neq \varnothing$ do
5: $\quad u =$ EXTRACT-MIN(Q)
6: $\quad S = S \bigcup \{u\}$
7: $\quad$ for each vertex $v \in G.Adj[u]$ do
8: $\quad\quad$ RELAX(u, v, w)

INITIALIZE-SINGLE-SOURCE(G, s)

1: for each vertex $v \in G.V$ do
2: $\quad v.d = \infty$
3: $\quad v.\pi = NIL$
4: $s.d = 0$

该算法的运行过程例如图7.3所示。

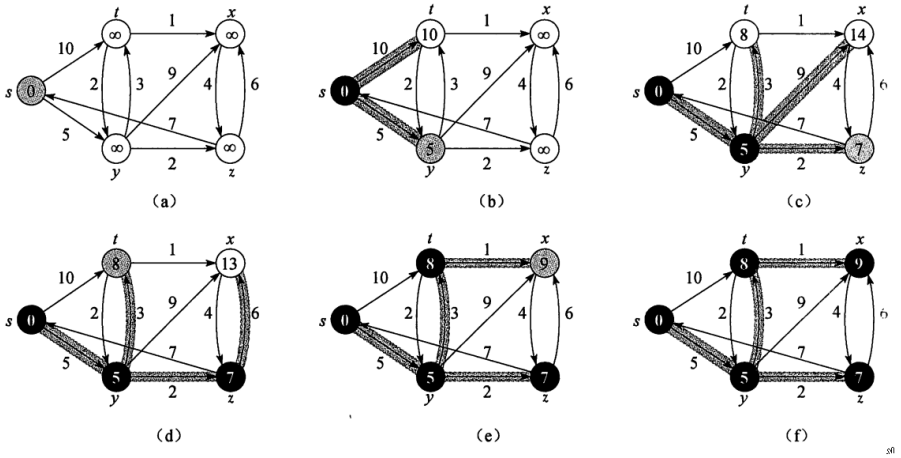

Figure 7.3: Dijkstra 算法示例

接下来我们证明这个算法是正确的，该算法的正确性依赖于贪心选择性质：当结点 v 被放入集合 S 时，$v.d = \delta(s, v)$。我们使用反证法来证明，如果此时 $v.d > \delta(s, v)$，那么由收敛性质，将来会有一条边被松弛 (u, v)，使得 $v.d$ 变小，即 u 还没有被从 Q 中取出来。因为此时 v 是 Q 中 d 值最小的，所以有 $v.d \leq u.d$，现在我们考察对边 (u, v) 的松弛操作，如果要使得 $v.d$ 变小，必须有 $v.d > u.d + w(u, v) > u.d$，这与 $v.d \leq u.d$ 相矛盾。因此当一个结点 u 被加入集合 S 后，那么 $u.d$ 就是从 s 走到 u 的最短路径距离。

最终，我们便可以通过数学归纳法来证明 Dijkstra 的正确性。该算法的时间复杂度为 $\Theta(|V| \lg |V| + |E|)$。

7.5.3 Bellman-Ford 算法

Bellman-Ford 算法可以解决一般情况下的最短路径问题，该算法可以解决边的权值存在负值情况的最短路径问题。首先，我们可以估计一个最短路径的上界，这个上界非常平凡，即 $|V| - 1$，然后我们可以由路径松弛性质导出一个算法，我们只要把每条边都松弛 $|V| - 1$ 次即可，这样我们就可以保证从 s 到每个结点的最短路径上的每一条边都被松弛过了一次。由此我们给出 Bellman-Ford 算法，该算法如果返回 False，则表示图中包含一个负权环路。

BELLMAN-FORD(G, w, s)

```
1: INITIALIZE-SINGLE-SOURCE(G, s)
2: for i = 1 to |G.V| − 1 do
3:     for each edge (u, v) ∈ E do
4:         RELAX(u, v, w)
5: for each edge (u, v) ∈ E do
6:     if v.d > u.d + w(u, v) then
7:         return False
8: return True
```

该算法的时间复杂度是 $\Theta(|V||E|)$（此处用 Dijkstra 算法的例子跑一下 Bellman-Ford 算法）。

7.5.4 Floyd-Warshall 算法

接下来我们介绍 Floyd-Warshall 算法，这是一个利用动态规划策略求解任意点对之间的最短路径的算法。该算法设 $d^{(k)}(i, j)$ 代表从结点 v_i 走到结点 v_j 的中间结点选自 $\{v_1, v_2, \cdots, v_k\}$ 的最短路径长度，那么为了计算这个值，可以分为以下两种情况：

1. 从 v_i 走到 v_j 的中间结点选自 $\{v_1, v_2, \cdots, v_k\}$ 的最短路径中不包括 v_k，即这条最短路径的中间结点均选自 $\{v_1, v_2, \cdots, v_{k-1}\}$，那么此时解为 $d^{(k-1)}(i, j)$。

2. 从 v_i 走到 v_j 的中间结点选自 $\{v_1, v_2, \cdots, v_k\}$ 的最短路径中包括 v_k，那么可以分解为从 v_i 走到 v_k 的最短路径中的结点和从 v_k 走到 v_j 最短路径中的结点均选自 $\{v_1, v_2, \cdots, v_{k-1}\}$，那么此时的最短路径长度为 $d^{(k-1)}(i,k) + d^{(k-1)}(k,j)$。

于是我们可以得到如 FLOYD-WARSHALL 所示的算法，该算法的时间复杂度是 $\Theta(|V|^3)$。

FLOYD-WARSHALL(W)

1: $n = W.rows$
2: $D^{(0)} = W$
3: for $k = 1$ to n do
4: Let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
5: for $i = 1$ to n do
6: for $j = 1$ to n do
7: $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

为了构建最短路径，我们需要在动态规划的构成中记录选择的子空间，记 $\Pi_{ij}^{(k)}$ 为从 v_i 到 v_j 的最短路径中的结点选自 $\{v_1, v_2, \cdots, v_k\}$ 时 j 的前驱结点。初始情况下，我们记

$$\Pi_{ij}^{(0)} = \begin{cases} NIL, & i = j \ \text{ or } \ w_{ij} = \infty \\ i, & i \neq j \ \text{ and } \ w_{ij} < \infty \end{cases}$$

在递推的过程中，记

$$\Pi_{ij}^{(k)} = \begin{cases} \Pi_{ij}^{(k-1)}, & d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \Pi_{kj}^{(k-1)}, & d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

于是我们可以用 PRINT-SHORTEST-PATH 算法来打印最短路径。

PRINT-SHORTEST-PATH($\Pi^{(n)}, i, j$)

1: if $\Pi_{ij}^{(n)} \neq NIL$ then
2: PRINT-SHORTEST-PATH($\Pi^{(n)}, i, \Pi_{ij}^{(n)}$)
3: Print($\Pi_{ij}^{(n)}$)

7.6 最小生成树

这一小节中我们将介绍求解无向图上的最小生成树的两种算法，这两种算法都基于贪婪策略。无向图上的最小生成树是指在一个无向图中选取 $|V| - 1$ 条边，使得这 $|V| - 1$ 条边能够将原图连接成一个连通图，同时总的边的权值应该是最小的。

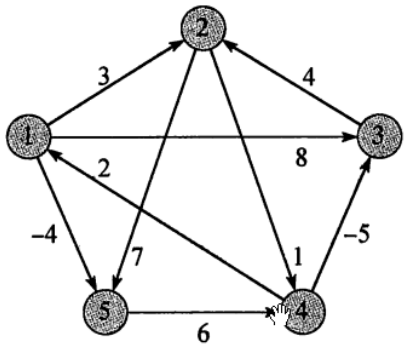

Figure 7.4: Floyd-Warshall 算法的示例图

$$
D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad \Pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}
$$

$$
D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad \Pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}
$$

$$
D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad \Pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}
$$

$$
D^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad \Pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}
$$

$$
D^{(4)} = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \quad \Pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}
$$

$$
D^{(5)} = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \quad \Pi^{(5)} = \begin{bmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}
$$

Figure 7.5: Floyd-Warshall 算法的示例流程

该问题满足最优子结构的性质，对于一棵最小生成树，那么它的一棵子树是它的节点构成的子图的最小生成树。否则就有一棵更小的最小生成子树可以替换掉当前子树，使得总的生成树的权值最小，因此满足最优子结构的性质。

7.6.1 Prim 算法

由于最小生成树满足最优子结构的性质，因此我们可以给出一个动态规划策略。首先我们先选择任意一个结点 u，设与结点 u 连接的边构成的边集为 $E = \{e_1, e_2, \cdots, e_n\}$，那么最小生成树的解空间可以划分为以下几个子空间:（1）结点 e_1 在最小生成树中，连接着 u 和 $V - \{u\}$ 构成的最小生成树。（2）结点 e_2 在最小生成树中，连接着 u 和 $V - \{u\}$ 构成的最小生成树。$\cdots$（n）结点 e_n 在最小生成树中，连接着 u 和 $V - \{u\}$ 构成的最小生成树。于是，原图的最小生成树就是这些子空间求得的生成树中最小的那个。

在划分子空间的过程中我们可以发现，最小生成树一定存在于 $w(e), e \in E$ 最小的那个子空间中。于是我们便可以写出一个贪婪策略的算法了，这就是 Prim 算法，如 PRIM 所示。

PRIM(G, w, r)

```
 1: for each u ∈ G.V do
 2:     u.key = ∞
 3:     u.π = NIL
 4: r.key = 0
 5: Q = G.V
 6: while Q ≠ ∅ do
 7:     u = EXTRACT-MIN(Q)
 8:     for each v ∈ G.Adj[u] do
 9:         if v ∈ Q and w(u, v) < v.key then
10:             v.π = u
11:             v.key = w(u, v)
```

该算法的时间复杂度为 $\Theta(|V| \lg |V| + |E| \lg |V|) = \Theta(|E| \lg |V|)$，因为第 11 行的设置 *key* 隐含一个调整堆的操作。

7.6.2 Kruskal 算法

另一个算法是 Kruskal 算法，它的思想是替换边。假设我们有一棵生成树 T，此时我们从原图当中选取一条边 e，此时我们将这条边 e 加入这棵生成树一定生成一个环，如果我们能够从这个环里找到另一条边 e'，满足 $w(e') > w(e)$，那么我们就将 e' 删掉，就能够得到另一棵总权重更小的生成树，我们重复这个操作直到生成树的总权重无法下

降的时候，我们便得到了这个图的最小生成树。

下面我们来讨论这棵最小生成树的性质，首先，原图当中最小权重的边一定在这棵最小生成树中。否则我们就可以将这条边加入到最小生成树当中，然后破除由此形成的环即可。因此从构造性的角度来说，我们可以在算法的开始，选择最小权重的边加入到边集 S 当中。接下来我们来看剩余的边集中权重最小的那个，如果这个边和在集合 S 当中的边 “冲突” 了，那么这个边一定不在最小生成树当中，否则加入进来就会形成一个环，而在破除这个环的方式当中，去掉其它任何一条边都将导致生成树的总权值变大，因为其它任何一条边都比这条边具有更小的权重。因此，如果这条边不与 S 当中的边冲突，我们就加入到 S 当中，否则就丢弃它，反复的进行这个操作直到遍历完所有的边，我们就能够得到 G 的最小生成树了，这就是 Kruskal 算法，如 KRUSKAL 所示。

KRUSKAL(G, w)

1: $A = \varnothing$
2: for each vertex $v \in G.V$ do
3: MAKE-SET(v)
4: sort $G.E$ by their weights
5: for each $(u, v) \in G.E$ taken by their weights do
6: if FIND-SET$(u) \neq$ FIND-SET(v) then
7: $A = A \bigcup \{(u, v)\}$
8: UNION(u, v)
9: return A

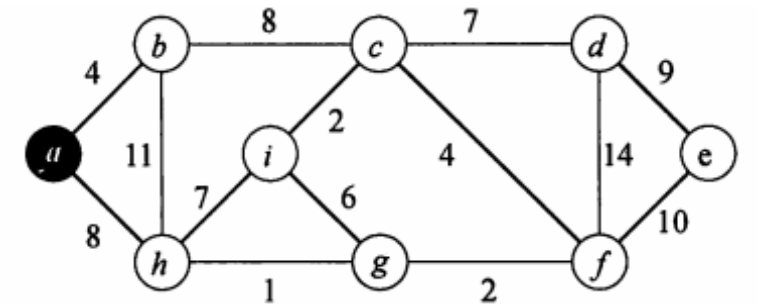

Figure 7.6: 最小生成树示例图

该算法要用到并查集，时间复杂度为 $\Theta(|E| \lg |E|)$，因为要对边集排序。