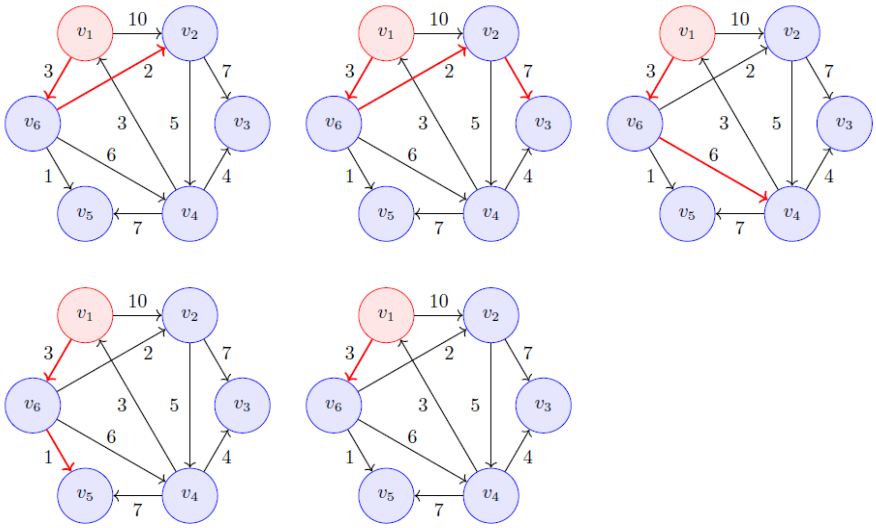
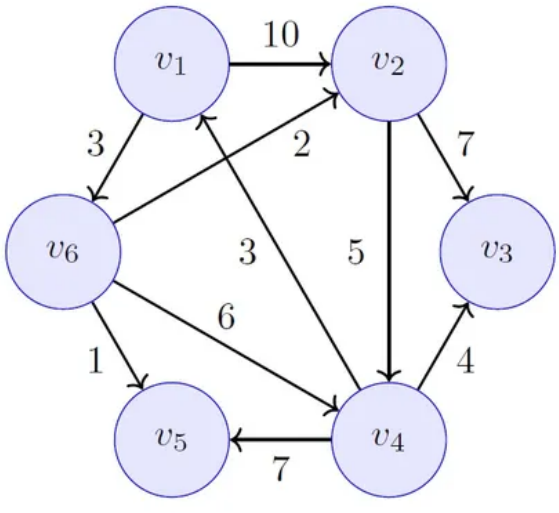


Dijkstra 算法，是由荷兰计算机科学家 Edsger Wybe Dijkstra 在1956年发现的算法，戴克斯特拉算法使用类似广度优先搜索的方法解决**赋权图的单源最短路径问题。**Dijkstra 算法原始版本仅适用于找到两个顶点之间的最短路径，后来更常见的变体固定了一个顶点作为源结点然后找到该顶点到图中所有其它结点的最短路径，产生一个最短路径树。本算法每次取出未访问结点中距离最小的，用该结点更新其他结点的距离。需要注意的是绝大多数的Dijkstra 算法不能有效处理带有**负权边**的图。

下面，我们就从一个赋权的有向图为例开始解释Dijkstra 算法。

设一个赋权有向图 $G=(V,E,W)$。其中的每条边 $e_{i,j}:=\{v_i,v_j\}$ 的权值为一个非负的实数 $w_{i,j}(e_{i,j})$，**该权值表示从顶点 v_i 到顶点 v_j 的距离。**并设一单源点 $s\in V$。现在我们的任务是：找出从源点 s 出发，到 $V\setminus\{s\}$ 中所有的节点的最短路径。

我们来看一个具体的例子：

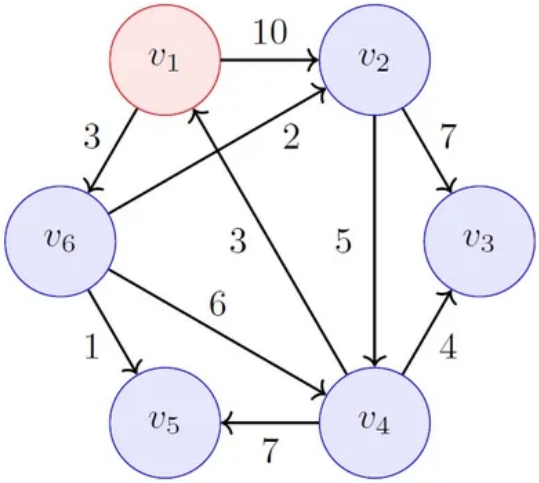


图片1：例图。

这是一个具有 6 个顶点的赋权有向图，其顶点集合为 $V=\{v_1,v_2,v_3,v_4,v_5,v_6\}$，其权值分别为：

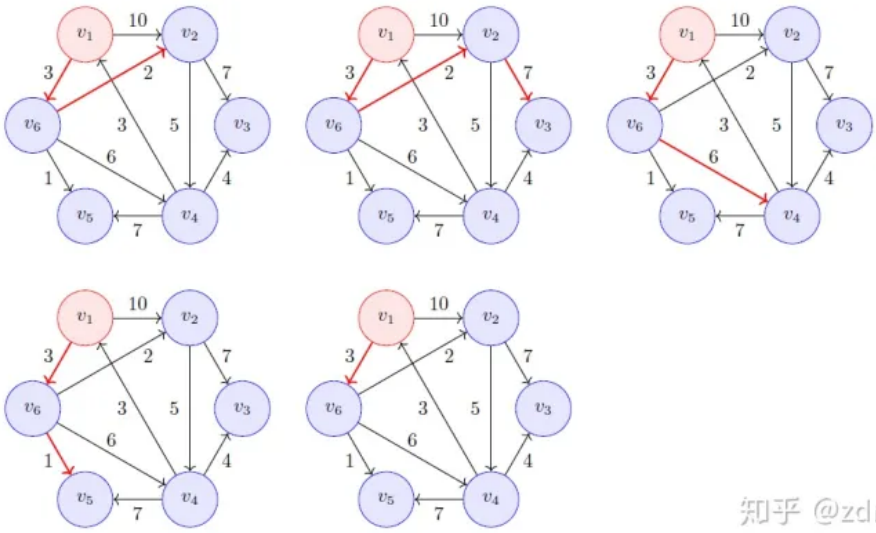
$$\begin{aligned} w_{1,2}\left(e_{1,2}\right)&=10 \quad w_{2,3}\left(e_{2,3}\right)=7\\ w_{4,3}\left(e_{4,3}\right)&=4 \quad w_{4,5}\left(e_{4,5}\right)=7\\ w_{6,5}\left(e_{6,5}\right)&=1 \quad w_{1,6}\left(e_{1,6}\right)=3\\ w_{6,2}\left(e_{6,2}\right)&=2 \quad w_{4,1}\left(e_{4,1}\right)=3\\ w_{2,4}\left(e_{2,4}\right)&=5 \quad w_{6,4}\left(e_{6,4}\right)=6 \end{aligned} \tag{1}$$

现在我们选定 v_1 为原点 s：



图片2：选择v_1作为原点s。

则从源点 v_1 出发，到 $V \setminus \{v_1\}$ 中所有顶点的最短路径分别为：



图片3：V中的源点s(v_1)到V中其余点的最短路径。

即：

$$\textbf{short}\,[2] : v_1 \to v_6 \to v_2 \Rightarrow \textbf{short}\,[2] = w_{1,6} + w_{6,2} = 5 \tag{2}$$
$$\textbf{short}\,[3] : v_1 \to v_6 \to v_2 \to v_3 \Rightarrow \textbf{short}\,[3] = w_{1,6} + w_{6,2} + w_{2,3} = 12 \tag{3}$$
$$\textbf{short}\,[4] : v_1 \to v_6 \to v_4 \Rightarrow \textbf{short}\,[4] = w_{1,6} + w_{6,4} = 9 \tag{4}$$
$$\textbf{short}\,[5] : v_1 \to v_6 \to v_5 \Rightarrow \textbf{short}\,[5] = w_{1,6} + w_{6,5} = 4 \tag{5}$$
$$\textbf{short}\,[6] : v_1 \to v_6 \Rightarrow \textbf{short}\,[6] = w_{1,6} = 3 \tag{6}$$

其中，$\textbf{short}[\bullet]$ 表示从源点 $s(v_1)$ 出发，到 $V \setminus \{v_1\}$ 中的顶点 $\bullet$ 的最短路径。

> 注：**最短路径可以理解为所有可能的路径中总权和最小的那一条路径。**举一个再简单不过的例子：你开车从城市 A 到城市 B，假设有很多条路可以走，最短的那条路就是最短路径，总权和可以理解为总的公里数。

以上是我们通过观察和计算比对出来的最短路径，下面我们就来看看Dijkstra 算法是如何帮助我们找到这些所有的最短路径的。

在开始之前，有几个概念需要明确一下。

- 定义一个集合 S，如果集合 $V \setminus \{s\}$ 中的某个顶点 v_i 在集合 S 中了，那么就说明从源点 s 到顶点 $v_i \in V \setminus \{s\}$ 的最短路径已经被找到，而在算法一开始的时候，集合 S 中只有源点 s。即：

$$S := \{v_i \in V : \text{the shortest path of vertex } v_i \text{ has been found}\} \tag{7}$$

而且，当且仅当 $S = V$ 的时候算法执行完毕。此时顶点集 V 中的所有元素都被放进了集合 S 种，也就是说除了源点以外的所有从源点出发到其余所有顶点的最短路径已被找到。

> 注：当然了，你也可以认为源点到自己本身的最短路径也被找到了。对于任意一个**无自环的**源点，它到自己本身的最短路径都是 0。

- 下面这个概念可能稍微有些抽象，不过没有关系，这里理解不了的话我们一会讲例子的时候会进行具体说明。这个概念叫做**从源点 s 到顶点 $v_i \in V$（一开始 $v_i \notin S$）的相对于集合 S 的最短路径。** 即从源点 s 到顶点 $v_i \in V$ 的路径**中间**只能经过已经包含在集合 S 中的顶点，而不能经过其余的还未在集合 S 中的顶点。而这个相对于集合 S 的最短路径的长度我们记作：

$$\mathbf{dist}[s, v_i] \tag{8}$$

而我们之前的 $\mathbf{short}[\bullet]\ (\Leftrightarrow \mathbf{short}[s, v_i])$ 表示的是**全局的**从源点 s 到顶点 $v_i \in V \setminus \{s\}$ 的最短路径，这个最短路径没有限制“必须在路径中间只能经过已经包含在集合 S 中的顶点”，这个全局的最短路径才是我们要的最终解。所以，一般有关系：

$$\mathbf{dist}[s, v_i] \geq \mathbf{short}[s, v_i] \tag{9}$$

而我们的Dijkstra 算法要做的就是通过不断计算 $\mathbf{dist}[s, v_i]$ 进而不断的扩充集合 S，当集合 S 不断被扩充的时候，相对于集合 S 的最短路径会越来越短，直到 v_i 入集合 S 之时，此时我们便得到了 $\mathbf{short}[s, v_i]$，且此时有 $\mathbf{dist}[s, v_i] = \mathbf{short}[s, v_i]$。下面我们来看看算法的设计思想：

> **输入**：赋权有向图 $G = (V, E, W)$，　$V = \{v_1, v_2, \ldots, v_n\}$，　$s := v_1$。
> **输出**：从源点 s 到所有的 $v_i \in V \setminus \{s\}$ 的最短路径。
> **1.** 初始 $S = \{v_1\}$；
> **2.** 对于 $v_i \in V - S$，计算 $\mathbf{dist}[s, v_i]$；
> **3.** 选择 $\mathbf{min}_{v_j \in V}\ \mathbf{dist}[s, v_i]$，并将这个 v_j 放进集合 S 中，更新 $V - S$ 中的顶点的 $\mathbf{dist}$ 值；
> **4.** 重复 1，直到 $S = V$。

然后是Dijkstra 算法的伪码：

Algorithm : Dijkstra

Input : Directed graph $G = (V, E, W)$ with weight

Output : All the shortest paths from the source vertex s to every

$1 : S \leftarrow \{s\}$
$2 : \text{dist}[s, s] \leftarrow 0$
$3 : \textbf{for } v_i \in V - \{s\} \textbf{ do}$
$4 : \quad \text{dist}[s, v_i] \leftarrow w(s, v_i)$
$\quad (\text{when } v_i \text{ not found}, \text{dist}[s, v_i] \leftarrow \infty)$
$5 : \textbf{while } V - S \neq \varnothing \textbf{ do}$
$6 : \quad \text{find } \min_{v_j \in V} \text{dist}[s, v_i] \text{ from the set } V - S$
$7 : \quad S \leftarrow S \cup \{v_j\}$
$8 : \quad \textbf{for } v_i \in V - S \textbf{ do}$
$9 : \quad \quad \textbf{if } \text{dist}[s, v_j] + w_{j,i} < \text{dist}[s, v_i] \textbf{ then}$
$10 : \quad \quad \quad \text{dist}[s, v_i] \leftarrow \text{dist}[s, v_j] + w_{j,i}$

下面我们来解释一下这个伪码：

1 : 算法初始，将选择的源点 s 放进集合 S 中；

2 : 无自环的源点 s 到自己的最短路径为 0 ；

3 : 当顶点 v_i 不在集合 S 中时（此时集合 S 中仍只有源点 s ），开始进入循环；

4 : 将源点 s 与点 v_i 之间的权值赋给 $\text{dist}[s, v_i]$ 。由于是有向图，所以当源点 s 不指向任何其他集合 S 外的顶点时， $\text{dist}[s, v_i] = \infty$ 。可以理解为**此时**从源点 s 出发，**暂时**是达到不了 v_i 的。不过后来随着集合 S 的扩充，从源点 s 出发一定能到达所有的顶点。一会我们讲解例子时会出现这种情况。此时第一个 **for** 循环结束。

5 : 如果集合 $V - S$ 不是空集，则进入循环；

6 : 选出经过第一个 **for** 循环之后的，在集合 $V - S$ 中的，且相对于集合 S 的最短路径中距离最短的那个顶点 v_j ；

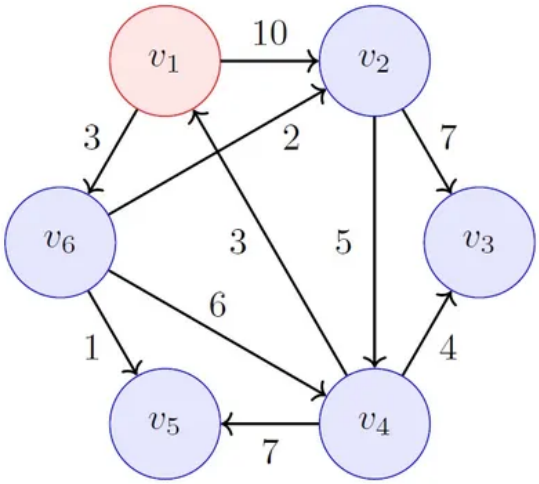
7 : 将这个顶点 v_j 并入集合 S ，从而达到扩充集合 S 的目的；

8 : 将顶点 v_i 并入集合 S 之后可能会对其他顶点相对于集合 S 的最短路的长度会有影响，所以进入内 **for** 循环对有影响的进行更新；

9 : 即如果从源点 s 到我们在第 6 步选出的顶点 v_j 的相对于集合 S 的最短路径的长度再加上顶点 v_j 到顶点 v_i 之间的距离 $w_{i,j}$ 还要小于源点 s 到顶点 v_i 的相对于集合 S 的最短路径的长度还要短的话；

10 : 则将源点 s 到顶点 v_i 的相对于集合 S 的最短路径更新成源点 s 到我们在第 6 步选出的顶点 v_j 的相对于集合 S 的最短路径再加上顶点 v_j 到顶点 v_i 之间的权值 $w_{i,j}$ 。

下面我们开始讲例子，我们还是以图片1中的赋权有向图进行说明。



图片4

首先我们还是选择 v_1 为原点 s，那么在算法的开始，$S=\{v_1\}$。之后我们计算除了 v_1 以外的其余顶点到 $s=v_1$ 的距离 $\mathbf{dist}[v_2,v_1]\sim\mathbf{dist}[v_6,v_1]$，即寻找所有的除了 v_1 以外的所有顶点相对于集合 S 的最短路，即从 v_1 出发，到达所有顶点且只允许通过顶点 v_1（因为此时集合 S 中只有 v_1 这一个元素）的最短路径。这是我们的算法中的第一个 **for** 循环在做的事情。这时候我们发现想要**只通过**顶点 v_1 而到达顶点 v_4,v_3,v_5 都是不可能的，所以我们有：

$$\begin{aligned}\mathbf{dist}[v_1,v_2]=w_{1,2}=10 \quad \mathbf{dist}[v_1,v_3]=\infty\\ \mathbf{dist}[v_1,v_4]=\infty \quad \mathbf{dist}[v_1,v_5]=\infty\\ \mathbf{dist}[v_1,v_6]=w_{1,6}=3 \quad \mathbf{dist}[v_1,v_1]=0\end{aligned}\tag{11}$$

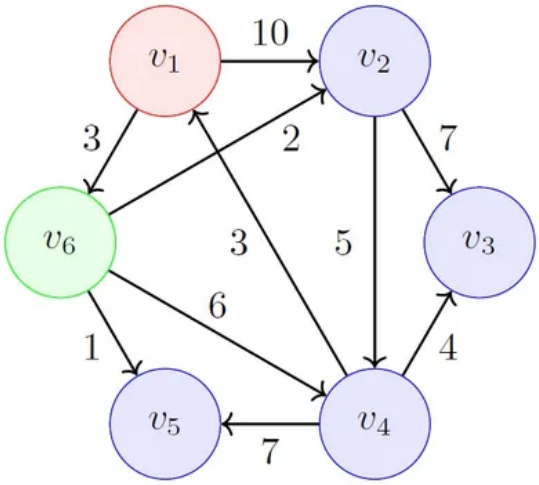
而 v_4,v_3,v_5 就是算法中所说的暂时到达不了的顶点了。现在算法的前四步已经结束了，现在开始第五步检验集合 $V-S$ 是否是空集，这里显然不是，这里：

$$V-S=\{v_2,v_3,v_4,v_5,v_6\}\tag{12}$$

现在进行第六步。第六步是选出经过第一个 **for** 循环之后的，在集合 $V-S$ 中的，且相对于集合 S 的最短路径中距离最短的那个顶点 v_j。那我们看看在式 (11) 中那个顶点距离源点 v_1 最短就好了，显然是 v_6，所以，我们这里选择的 $v_j=v_6$。

那么第七步就是将 v_6 放进集合 S 中了。此时集合 $S=\{v_1,v_6\}$。这就是说明从源点 s 出发，到顶点 v_6 的最短路径已经被找到了。

下面我用绿色表示被放入集合 S 中的顶点：



图片5：顶点v_6被加进集合S中。

v_1 的颜色我就不变了，因为它一直都在集合 S 中。此时：

$$S = \{v_1, v_6\} \tag{13}$$

这就说明下次在找相对于集合 S 的最短路径的时候 S 中就有两个点可以被通过了，这样也许就会使得一些原来到达不了的顶点由于可以多经过一个点而到达，这也就是算法中所说的当我将一个新的顶点并入集合 S 之后，其他的在集合 $V-S$ 以外的顶点的相对于集合 S 的最短路径的长度可能会发生改变，因为有些原来暂时到达不了的顶点现在可以到达了。具体的来讲，我们有：

$$\begin{aligned} \mathbf{dist}[v_1, v_2] &= \mathbf{dist}[v_1, v_6] + w_{6,2} = 5 & \mathbf{dist}[v_1, v_3] &= \infty \\ \mathbf{dist}[v_1, v_4] &= \mathbf{dist}[v_1, v_6] + w_{6,4} = 9 & \mathbf{dist}[v_1, v_5] &= \mathbf{dist}[v_1, v_6] + w_{6,5} = 4 \\ \mathbf{dist}[v_1, v_6] &= w_{1,6} = 3 & \mathbf{dist}[v_1, v_1] &= 0 \end{aligned} \tag{14}$$

这个更新步骤我也来详细是说一下，这是算法第八到第十步所做的事情。比如 $\mathbf{dist}[v_1, v_2]$，一开始在集合 S 中只有源点 v_1，而找到 v_2 相对于集合 S 的最短路径只能通过顶点 v_1，这样我们在式 (11) 中所得到 $\mathbf{dist}[v_1, v_2] = w_{2,1} = 10$。但是当顶点 v_6 也进入到集合 S 之后我们再找 v_2 相对于集合 S 的最短路径时就可以先通过顶点 v_1 然后到顶点 v_6，最后再到 v_2。现在这两种走法都可以，但是算法究竟选择哪种算法还是要判断哪种走法距离最短，即比较：

$$\mathbf{dist}[v_1, v_2], \quad \mathbf{dist}[v_1, v_6] + w_{6,2} \tag{15}$$

之间的大小关系，谁小算法就选择谁。经过比较发现：

$$\mathbf{dist}[v_1, v_2] = 10 > \mathbf{dist}[v_1, v_6] + w_{6,2} = 5 \tag{16}$$

所以选择后者。再比如原来达到不了的 v_4，现在由于集合 S 中多了顶点 v_6 变得可以达到了，即：

$$\mathbf{dist}[v_1, v_4] = \infty \gg \mathbf{dist}[v_1, v_6] + w_{6,4} = 9 \tag{17}$$

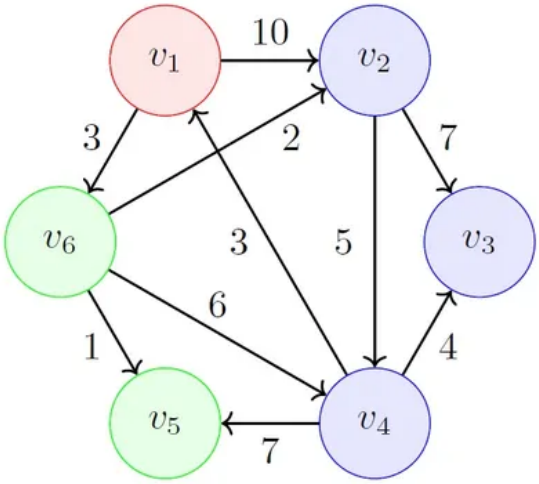
所以算法肯定选择后者。不过此时算法也没得可选，先要到达顶点 v_4 就必须走这条路。

其余发生变化的顶点分析类似，大家可以自己试试。

现在算法从头到尾被执行了一遍了，然后我们回到第五步判断 **while** 循环的条件还是否为真，此时：

$$V - S = \{v_2, v_3, v_4, v_5\} \neq \varnothing \tag{18}$$

所以再执行 **while** 循环，由第六步从式 (14) 中选择出属于集合 $V-S$，且相对于集合 S 的最短路径中距离最短的那个顶点为 v_j，所以，这里我们选择的 $v_j = v_5$。然后第七步 将 v_5 放进集合 S。此时，集合 $S = \{v_1, v_6, v_5\}$：



图片6：v_5被放进集合S中。

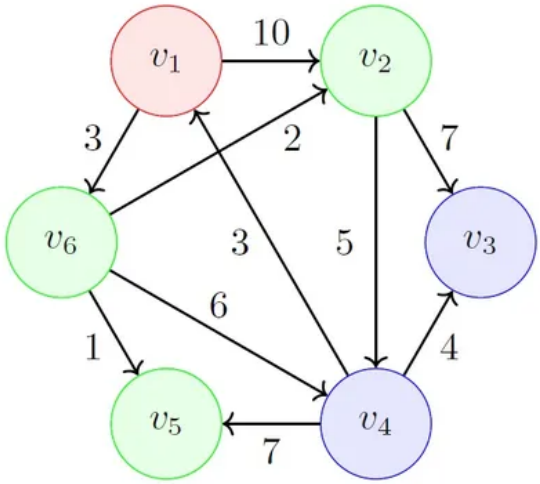
此时我们有：

$$\begin{aligned} \mathbf{dist}[v_1, v_2] &= \mathbf{dist}[v_1, v_6] + w_{6,2} = 5 & \mathbf{dist}[v_1, v_3] &= \infty \\ \mathbf{dist}[v_1, v_4] &= \mathbf{dist}[v_1, v_6] + w_{6,4} = 9 & \mathbf{dist}[v_1, v_5] &= \mathbf{dist}[v_1, v_6] + w_{6,5} = 4 \\ \mathbf{dist}[v_1, v_6] &= w_{1,6} = 3 & \mathbf{dist}[v_1, v_1] &= 0 \end{aligned} \tag{19}$$

可见这次没有发生更新，且此时的：

$$V - S = \{v_2, v_3, v_4\} \neq \varnothing \tag{20}$$

所以再执行 **while** 循环，由第六步从式 (19) 中选择出属于集合 $V-S$，且相对于集合 S 的最短路径中距离最短的那个顶点为 v_j，所以，这里我们选择的 $v_j = v_2$。然后第七步 将 v_2 放进集合 S。此时，集合 $S = \{v_1, v_6, v_5, v_2\}$：



图片7：v_2被放进集合S中。

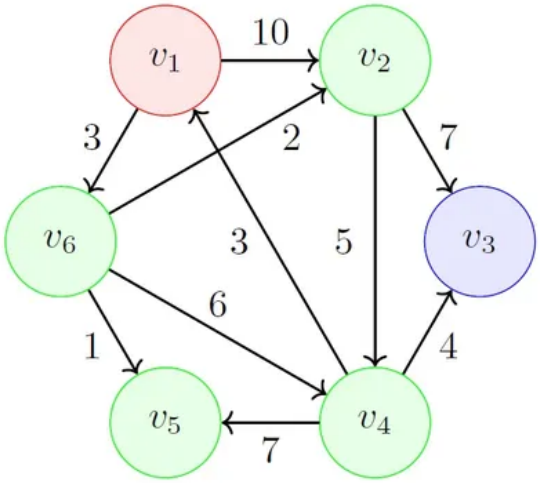
此时我们有：

$$\begin{aligned} \mathbf{dist}[v_1, v_2] &= \mathbf{dist}[v_1, v_6] + w_{6,2} = 5 & \mathbf{dist}[v_1, v_3] &= \mathbf{dist}[v_1, v_6] + w_{6,2} + w_{2,3} = 12 \\ \mathbf{dist}[v_1, v_4] &= \mathbf{dist}[v_1, v_6] + w_{6,4} = 9 & \mathbf{dist}[v_1, v_5] &= \mathbf{dist}[v_1, v_6] + w_{6,5} = 4 \\ \mathbf{dist}[v_1, v_6] &= w_{1,6} = 3 & \mathbf{dist}[v_1, v_1] &= 0 \end{aligned} \tag{21}$$

可见这次在顶点 v_3 处发生了更新（至于为什么 $\mathbf{dist}[v_1, v_3] \neq 17$ 大家可以自己分析一下试试），且此时的：

$$V-S=\{v_3,v_4\}\neq\varnothing \tag{22}$$

所以再执行 **while** 循环，由第六步从式 (19) 中选择出属于集合 $V-S$，且相对于集合 S 的最短路径中距离最短的那个顶点为 v_j，所以，这里我们选择的 $v_j=v_4$。然后第七步 将 v_4 放进集合 S。此时，集合 $S=\{v_1,v_6,v_5,v_2,v_4\}$：



图片8：v_4被放进集合S中。

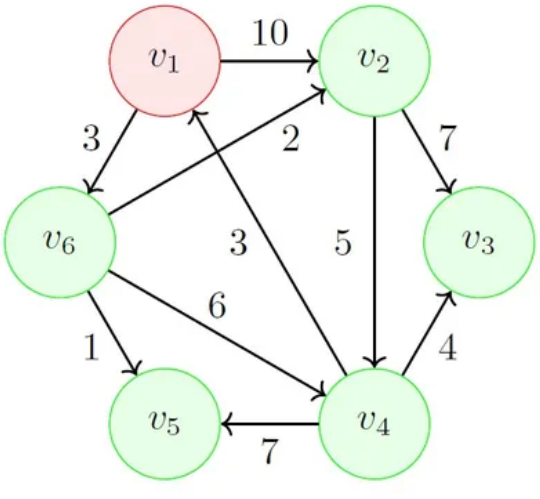
此时我们有：

$$\begin{gathered}\mathbf{dist}[v_1,v_2]=\mathbf{dist}[v_1,v_6]+w_{6,2}=5\quad \mathbf{dist}[v_1,v_3]=\mathbf{dist}[v_1,v_6]+w_{6,2}+w_{2,3}=12\\ \mathbf{dist}[v_1,v_4]=\mathbf{dist}[v_1,v_6]+w_{6,4}=9\quad \mathbf{dist}[v_1,v_5]=\mathbf{dist}[v_1,v_6]+w_{6,5}=4\\ \mathbf{dist}[v_1,v_6]=w_{1,6}=3\quad \mathbf{dist}[v_1,v_1]=0\end{gathered} \tag{23}$$

可见这次并未发生更新，且此时的：

$$V-S=\{v_3\}\neq\varnothing \tag{24}$$

所以再执行 **while** 循环，由第六步从式 (19) 中选择出属于集合 $V-S$，且相对于集合 S 的最短路径中距离最短的那个顶点为 v_j，所以，这里我们选择的 $v_j=v_3$ （至剩下 v_3 可以被选择了）。然后第七步 将 v_3 放进集合 S。此时，集合 $S=\{v_1,v_6,v_5,v_2,v_4,v_3\}$：



图片9：v_4被放进集合S中。

此时我们有：

$$\begin{gathered}\mathbf{dist}[v_1, v_2] = \mathbf{dist}[v_1, v_6] + w_{6,2} = 5 \quad \mathbf{dist}[v_1, v_3] = \mathbf{dist}[v_1, v_6] + w_{6,2} + w_{2,3} = 12 \\ \mathbf{dist}[v_1, v_4] = \mathbf{dist}[v_1, v_6] + w_{6,4} = 9 \quad \mathbf{dist}[v_1, v_5] = \mathbf{dist}[v_1, v_6] + w_{6,5} = 4 \\ \mathbf{dist}[v_1, v_6] = w_{1,6} = 3 \quad \mathbf{dist}[v_1, v_1] = 0\end{gathered} \tag{25}$$

这是最后一次了，且这次并未发生更新，且此时的：

$$V - S = \{\} = \varnothing \tag{26}$$

则不满足算法中的 **while** 循环的条件，循环结束，算法结束。显然，此时有 $V = S$ 。

最后我们来看一看Dijkstra 算法的时间复杂度。

Dijkstra 算法的时间复杂度是 $\mathcal{O}(nm)$ 。其中：

$$n = |V|, \quad m = |E| \tag{23}$$

分别为赋权有向图中的顶点个数和边的个数。Dijkstra 算法的时间复杂度是 $\mathcal{O}(nm)$ 是因为算法总共进行 $(n-1)$ 步，每一步选出一个具有最小 $\mathbf{dist}[s, \bullet]$ 值的顶点放入集合 S 中，需要 $\mathcal{O}(m)$ 的时间。

而选择基于堆实现的优先队列数据结构，可将Dijkstra 算法的时间复杂度降为 $\mathcal{O}(m \log n)$ 。